NAVAL POSTGRADUATE SCHOOL

Monterey, California

THESIS

A STOCHASTIC APPROACH TO SOLVING THE
2 1/2 DIMENSIONAL WEIGHTED REGION PROBLEM

by

Cary Allen Hilton, Jr.

June 1991

Thesis Advisor: Man-Tak Shing

Approved for public release; distribution is unlimited.

Unclassified

Security Classification of this page

REPORT DOCUMENTATION PAGE

1a Report Security Classification UNCLASSIFIED		1b Restrictive Markings		
2a Security Classification Authority		3 Distribution Availability of Report Approved for public release; distribution is unlimited.		
2b Declassification/Downgrading Schedule				
4 Performing Organization Report Number(s)		5 Monitoring Organization Report Number(s)		
6a Name of Performing Organization Naval Postgraduate School	6b Office Symbol *(If Applicable)* AS	7a Name of Monitoring Organization Naval Postgraduate School		
6c Address *(city, state, and ZIP code)* Monterey, CA 93943-5000		7b Address *(city, state, and ZIP code)* Monterey, CA 93943-5000		
8a Name of Funding/Sponsoring Organization	8b Office Symbol *(If Applicable)*	9 Procurement Instrument Identification Number		
8c Address *(city, state, and ZIP code)*		10 Source of Funding Numbers		

Program Element Number	Project No	Task No	Work Unit Accession No

11 Title *(Include Security Classification)* A STOCHASTIC APPROACH TO SOLVING THE $2\frac{1}{2}$ DIMENSIONAL WEIGHTED REGION PROBLEM

12 Personal Author(s) Cary A. Hilton, Jr.

13a Type of Report Master's Thesis	13b Time Covered From To	14 Date of Report *(year, month,day)* 1991, June	15 Page Count 128

16 Supplementary Notation The views expressed in this paper are those of the author and do not reflect the official policy or position of the Department of Defense or the U.S. Government.

17 Cosati Codes

Field	Group	Subgroup

18 Subject Terms *(continue on reverse if necessary and identify by block number)* Shortest path; simulated annealing; weighted region problem; two-and-one-half-dimensional path planning; anisotropic path planning

19 Abstract *(continue on reverse if necessary and identify by block number*

This thesis describes a method of computing a feasible path solution for the anisotropic weighted region problem. Heuristics are used to locate an initial starting solution. This starting solution is iteratively improved using a golden ratio search to produce a solution within a specified tolerance. The path solution is then randomly perturbed or detoured through different region frontiers, and the golden ratio search is again applied. These random detours are controlled by a process known as simulated annealing, which determines the number of detours made and decides whether to accept or reject each path solution. Better solutions are always accepted and worse solutions are accepted based on a probability distribution. Accepting worse solutions allows an opportunity to escape from a local minimum condition and continue the search for the optimal path. Since an exhaustive search is not performed, the globally optimal path may not be found, but a feasible path can be found with this method.

20 Distribution/Availability of Abstract [✓ X] unclassified/unlimited ☐ same as report ☐ DTIC users	21 Abstract Security Classification Unclassified	
22a Name of Responsible Individual Man-Tak Shing	22b Telephone *(Include Area code)* (408) 646-2634	22c Office Symbol CS/Sh

DD FORM 1473, 84 MAR 83 APR edition may be used until exhausted security classification of this page

All other editions are obsolete Unclassified

A Stochastic Approach to Solving the 2 1/2 Dimensional Weighted Region Problem

by

Cary Allen Hilton, Jr.
Captain, United States Army
B.S., University of Southern Mississippi, 1980

Submitted in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE IN COMPUTER SCIENCE

From the

ABSTRACT

This thesis describes a method of computing a feasible path solution for the anisotropic weighted region problem. Heuristics are used to locate an initial starting solution. This starting solution is iteratively improved using a golden ratio search to produce a solution within a specified tolerance. The path solution is then randomly perturbed or detoured through different region frontiers, and the golden ratio search is again applied. These random detours are controlled by a process known as simulated annealing, which determines the number of detours made and decides whether to accept or reject each path solution. Better solutions are always accepted and worse solutions are accepted based on a probability distribution. Accepting worse solutions allows an opportunity to escape from a local minimum condition and continue the search for the optimal path. Since an exhaustive search is not performed, the globally optimal path may not be found, but a feasible path can be found with this method.

H53385
C.1

TABLE OF CONTENTS

ACKNOWLEDGEMENTS

I wish to thank Professor Man-Tak Shing for lending his skillful guidance, clever ideas, and detailed knowledge to this work. Professor Neil C. Rowe also provided valuable guidance and support. I am honored to have them associated with this thesis.

I also wish to thank Major Jerry A. Crane, who made molehills out of many mountains, and Hania La Born, whose typing skills were absolutely essential to the completion of this project.

I. INTRODUCTION

A. GENERAL BACKGROUND

Military applications of autonomous vehicles are twofold: to delegate certain repetitive tasks to machines, allowing more efficient use of scarce human resources, and to avoid exposing humans to hazardous duties that could be satisfactorily performed by automated machines. An autonomous vehicle could transport supplies from a rear area support facility to a front line unit. It could also perform surveillance and measurement functions in areas contaminated by chemical or radiological agents. To accomplish these tasks, a vehicle must be capable of examining terrain data and selecting a path which permits efficient travel from a start point to a given destination.

This process begins with high-level path planning, where the vehicle may take into account terrain, weather, concealment, and exposure to hostile forces in selecting the general route to take. One way to model this task is by using the **weighted region problem**.

B. THE WEIGHTED REGION PROBLEM

1. Definition

The weighted region problem accepts input from a two-dimensional Cartesian map which models terrain as convex polygonal regions. Each region is assigned a cost coefficient or weight which is the cost per unit distance travelled, relative to other regions in the map. The cost of traversing a region is calculated by multiplying the region's weight by the Euclidean distance travelled through the region. See Figure 1.

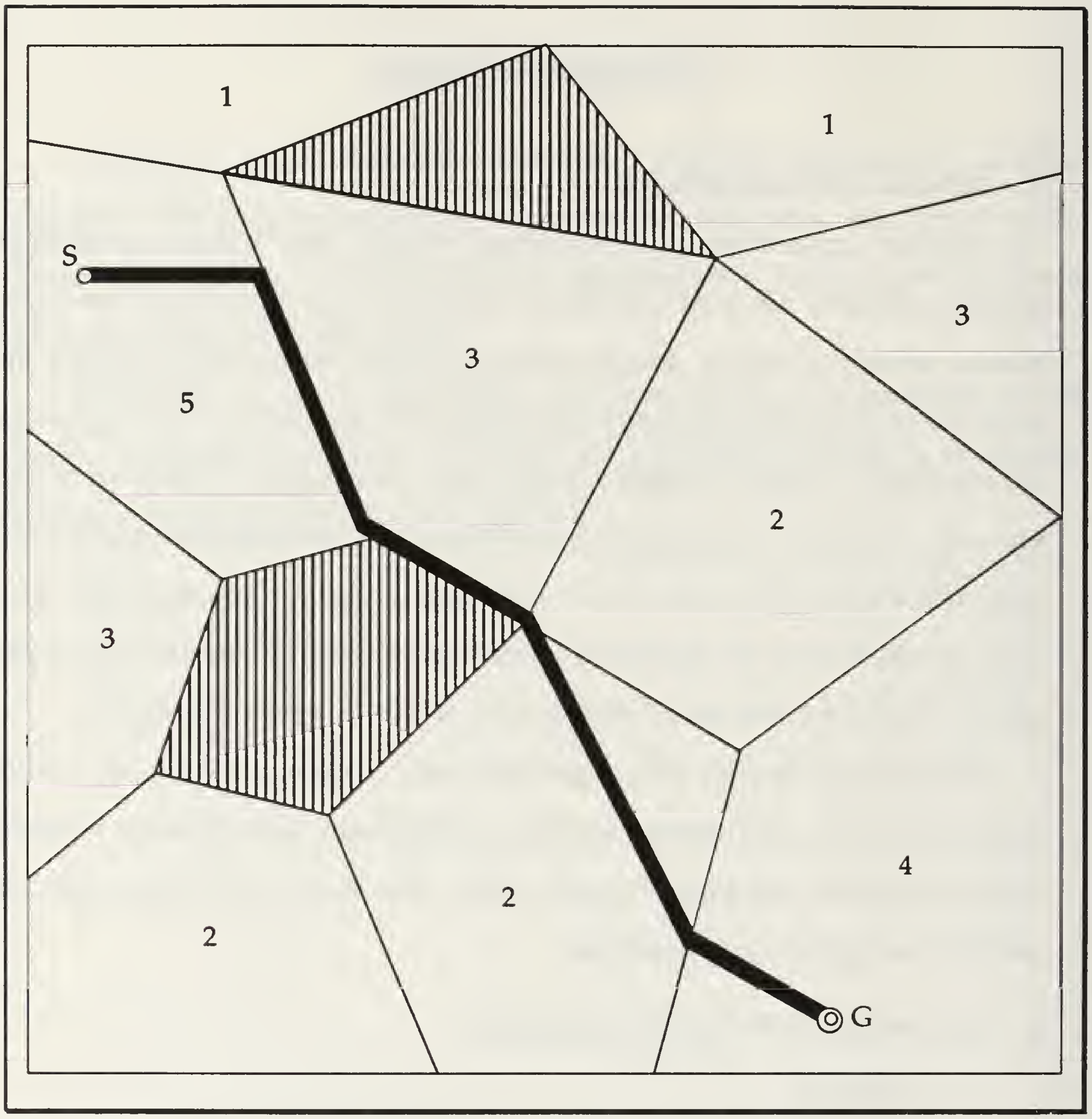


Figure 1. Weighted Region Problem

Given a start point and destination point in terms of map coordinates, the goal is to find the most efficient path from start to destination through the weighted regions.

An extension of the weighted region problem involves finding a path through sloped regions, where a ground vehicle's direction of travel is limited by its ability to climb steep hills and travel across sideslopes.

Researchers describe this path planning problem as the anisotropic weighted region problem. Anisotropic means that the direction of travel also influences the cost of traversing a region. For example, it requires more energy to travel uphill on a path than it does to travel downhill on the same path. A heading which would cause a vehicle to exceed its center-of-gravity limits can be avoided by assigning an infinite (or sufficiently large) cost to this path. Because of the added dimension of elevation, this problem is sometimes referred to as $2\frac{1}{2}$ dimensional path planning.

2. Problems to Overcome

An efficient path planning program is not trivial. This program must select each segment of the path by comparing the surface condition and slope of each region with the physical limitations of the vehicle. A method of partitioning the terrain must be determined. One way is to divide the terrain into a grid such that each grid square possesses homogeneous terrain with respect to surface composition, slope, and traversal cost. While this method may be relatively simple to implement, a large number of grid squares requires large memory allocation and the solution path may not be safe due to the "stair stepping" characteristics of moving from one grid square to a neighboring grid square.

In another technique, partitions are only made when necessary. A region is defined by homogeneous terrain, and is independent of size or shape. This approach reduces memory requirements, but complicates the search for the optimal path because the traditional "shortest path" graph search algorithms do not provide sufficient resolution to obtain an efficient path. Most implementations of this technique restrict the regions to convex polygons which simplifies the search and cost-calculating processes.

The vehicle must avoid areas where its power or stability limitations are exceeded. If not, the vehicle will find itself facing a hill which it cannot climb or a sideslope which will cause it to overturn. Paths which lead downhill may also be undesirable. If the destination's elevation is equal to or higher than the start point's elevation, then travelling downhill only increases the net elevation the vehicle must climb in order to reach its destination.

Once the program eliminates non-negotiable terrain regions from consideration, it must examine the remaining regions and select an acceptable path. These remaining regions may or may not be traversable, depending upon direction of travel. Again, the vehicle must avoid path headings that lead to excessively steep climbs or a potential rollover condition.

The calculations required to conduct an exhaustive search for an optimal path in $2\frac{1}{2}$ dimensional space grow exponentially with the number of regions in the search space. Accuracy is a function of resolution of the search space. Greater accuracy requires smaller grids or regions that provide a more accurate representation of the terrain, but increase the number of calculations necessary to investigate each segment of the search space (or map). This computational workload requires the use of a computer to speed up the process and determine the results in a reasonable amount of time.

Even with the aid of a computer, the time required to determine an optimal path may be excessive. Current algorithms do not find an optimal path through $2\frac{1}{2}$ dimensional space within acceptable time limits.

II. PREVIOUS WORK

A. ALGORITHMS

Several algorithms have been developed to solve the weighted region problem, including wavefront propagation [Ref. 1] and the continuous Dijkstra algorithm [Ref. 2]. The algorithms most impacting on this work are the systematic search by Ross and the stochastic approach by Kindl.

B. SYSTEMATIC SEARCH

Ross [Ref. 3] describes an algorithm which finds an optimal path through anisotropic weighted regions. He implemented his work with the Common Lisp programming language. Ross's algorithm used a map that was divided into homogeneous regions. The area within a region shares a common slope, orientation, surface composition and condition, stability and braking constraints.

He also defined a vehicle model which represents slope, climb, stability, and braking limits. He extensively described the trigonometric and physics equations used to determine which headings were allowable in each region.

Ross used an A* search algorithm to find feasible paths to investigate. He calls a sequence of regions that contains a feasible path from start to goal a window sequence. See Figure 2. For each feasible window sequence, the algorithm conducts an exhaustive search of all paths that lie within the permissible heading ranges.

Although this procedure finds the globally optimal path, the time required to find it renders the program impractical. Test results show it may require as much as 29 minutes to calculate a globally optimal path on a simple map containing 40 regions.

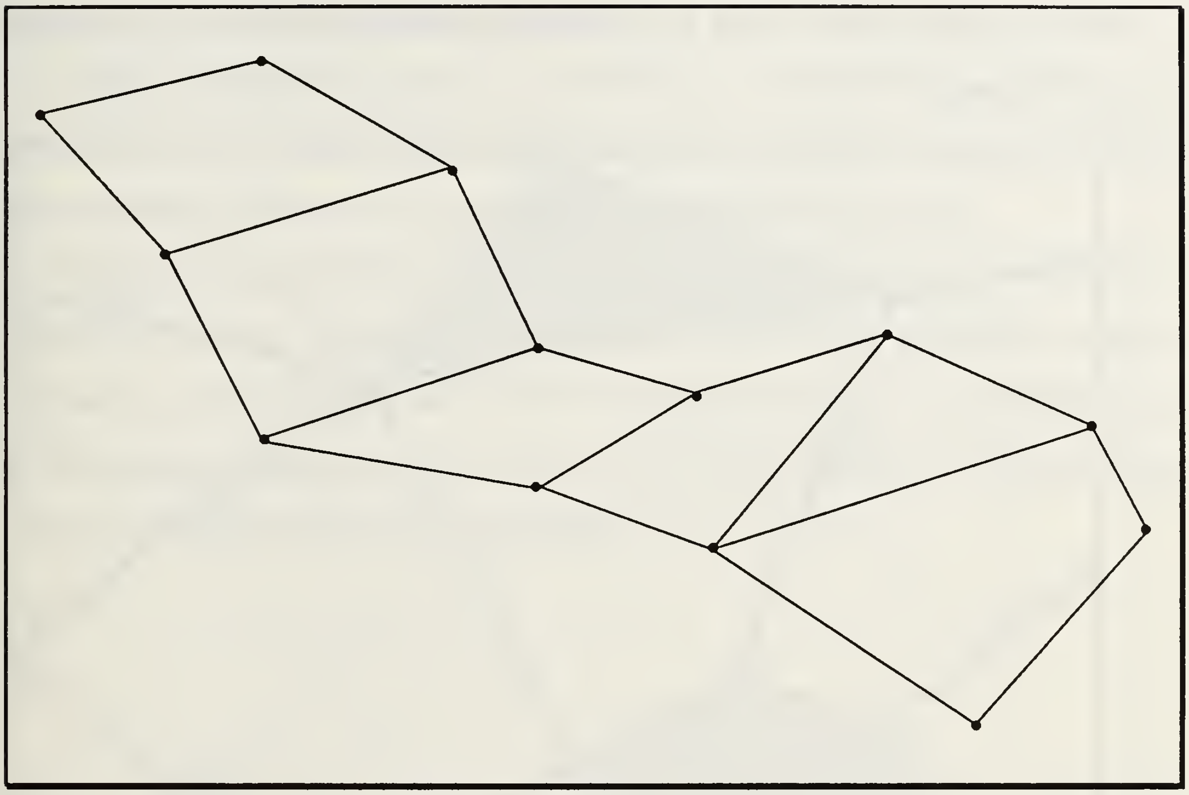


Figure 2. Window Sequence

C. STOCHASTIC APPROACH

Kindl [Ref. 4] introduced a stochastic approach to solving the isotropic weighted region problem. He implemented his work using the C++ and Prolog languages. His approach is based on a technique called simulated annealing. Like Ross, Kindl defined the regions on a map by edges and vertices. Kindl placed a node at the mid-point of each edge and defined a series of arcs which connect every node in a region. He calls this structure an edge dual graph. See Figure 3. Kindl then finds a feasible window sequence by performing an A* search on the edge dual graph. This path is the basis for finding a locally optimal path through the generated window sequence. The locally optimal path is found by iteratively applying golden ratio search to the path segments.

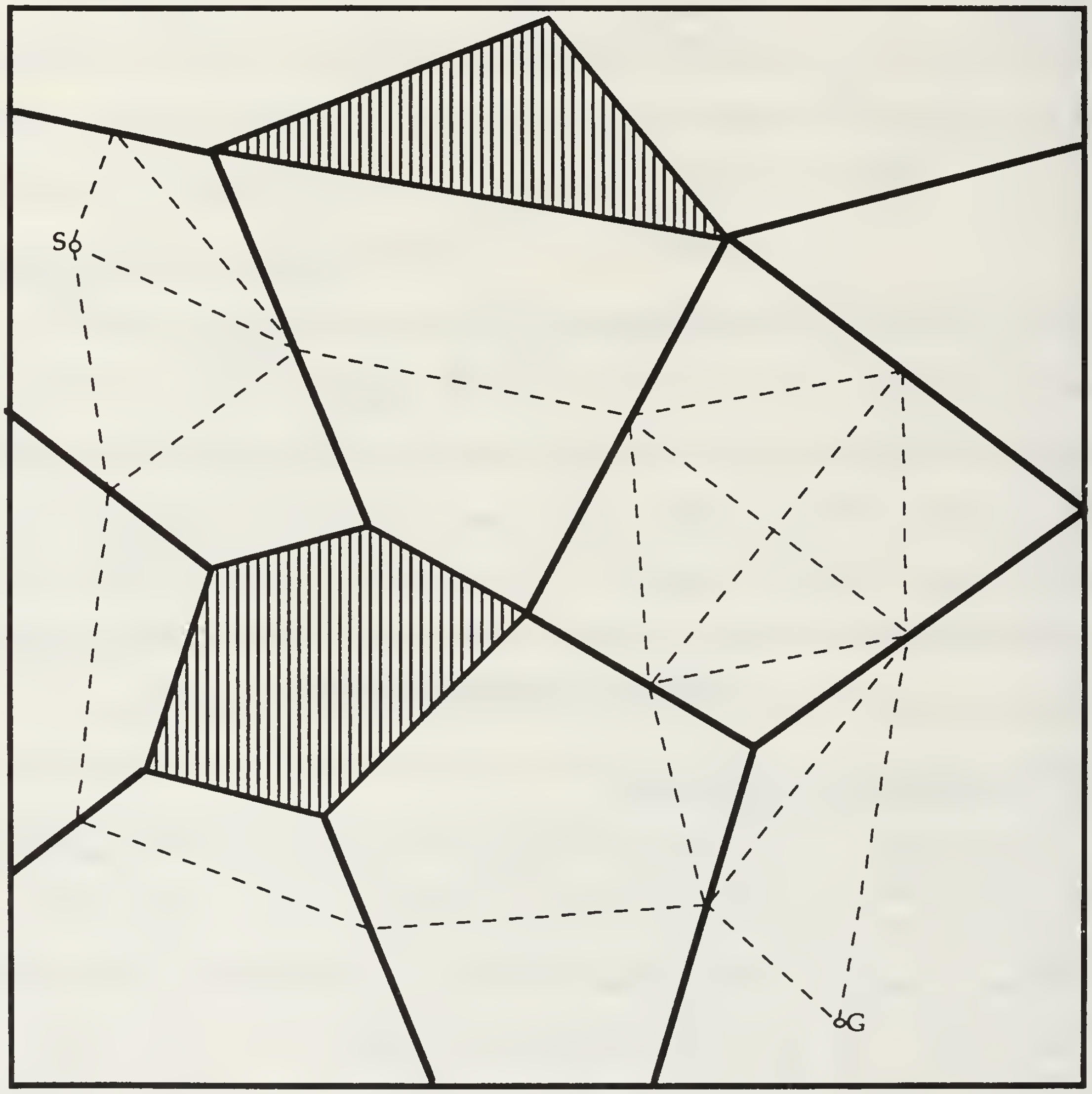


Figure 3. Edge Dual Graph

Kindl's program then uses path annealing to perturb the window sequence by randomly re-routing path sequences through neighboring regions in an attempt to find a better path.

Eligible neighboring regions are limited by a bounding ellipse which is an oval drawn around the start and goal. Regions or portions of regions outside the bounding ellipse are not considered by the annealing process.

While not guaranteed to find the globally optimal path, this procedure does find a near optimal path in much less time. Although Kindl's procedure produced excellent results, its scope is limited to two dimensional path planning. Extension of Kindl's work to $2\frac{1}{2}$ dimensional path planning will significantly broaden its applications.

III. GENERAL PROBLEM-SOLVING STRATEGY

To solve the anisotropic weighted region problem, we need vehicle and terrain models, geometric spatial reasoning functions, search functions, cost functions, and annealing control functions.

Given a vehicle and map, the general strategy is:

1. perform a search through the map regions to create a region list which may contain an initial path solution.
2. calculate necessary information and put these regions into a data structure called window list.
3. for each adjacent pair of regions in the window list, iteratively
   (a) determine heading ranges and cost functions.
   (b) perform a golden ratio search on each heading range combination and locate the optimal heading range combination.
   (c) perform a golden ratio search on the region crossing frontier defined by the optimal heading range combination and locate (within tolerance) the optimal crossing point.
4. if any crossing point shifted more than the specified displacement, go to 3.
5. if any path segment headings are impermissible, attempt to detour around them to find a valid path segment.
6. use random annealing in an attempt to find a shorter path.

These steps are described in greater detail in subsequent chapters.

IV. MODELING THE VEHICLE AND TERRAIN

A. VEHICLE

The vehicle model used in this work is a simplified version of those used by Ross and Rowe [Ref. 5] and Rowe and Kanayama [Ref. 6]. We are concerned with vehicle-specific heading limits for climbing, braking, and slideslope travel. See Figure 4. Given a vehicle's heading limits and the slope and orientation of a region, critical headings can be computed by the following formulas:

$$\text{critical-p} = \arccos((\tan \text{limit-p}) / (\tan \text{slope}))$$

$$\text{critical-s} = \arcsin((\tan \text{limit-s}) / (\tan \text{slope}))$$

$$\text{critical-b} = \arccos((\tan \text{limit-b}) / (\tan \text{slope}))$$

where limit-p, limit-s, limit-b are, respectively, the slope angle thresholds where the vehicle's uphill, sideslope, and braking limits occur.

Each critical angle has a *dual*, which with the critical angle defines a heading range. These dual headings are calculated as follows:

$$\text{dual-p} = -\text{critical-p}$$

$$\text{dual-s} = \pi - \text{critical-s}$$

$$\text{dual-b} = -\text{critical-b}$$

$$\text{critical-l} = -\text{critical-s}$$

$$\text{dual-l} = \text{critical-s} - \pi$$

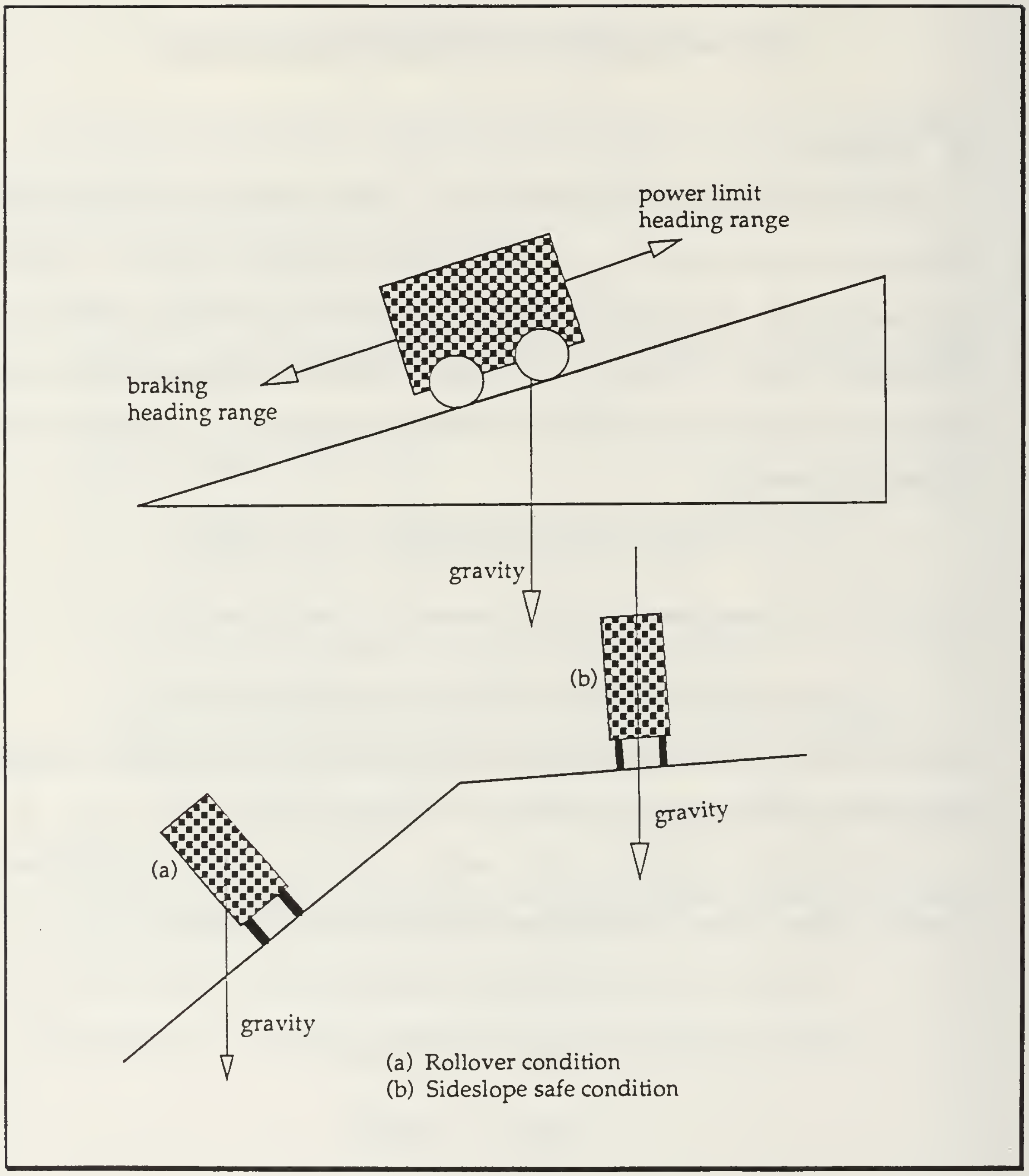


Figure 4. Vehicle Model

There are four ranges: power-limited, braking, and two sideslope ranges, one to the vehicle's left and the other to the right. If the vehicle's heading falls within one of these ranges, the corresponding condition applies. See Figure 5. For example, if the critical power-limit angle is 20° and its dual is 340°, a vehicle travelling at a heading of 355° is subject to the power-limited condition.

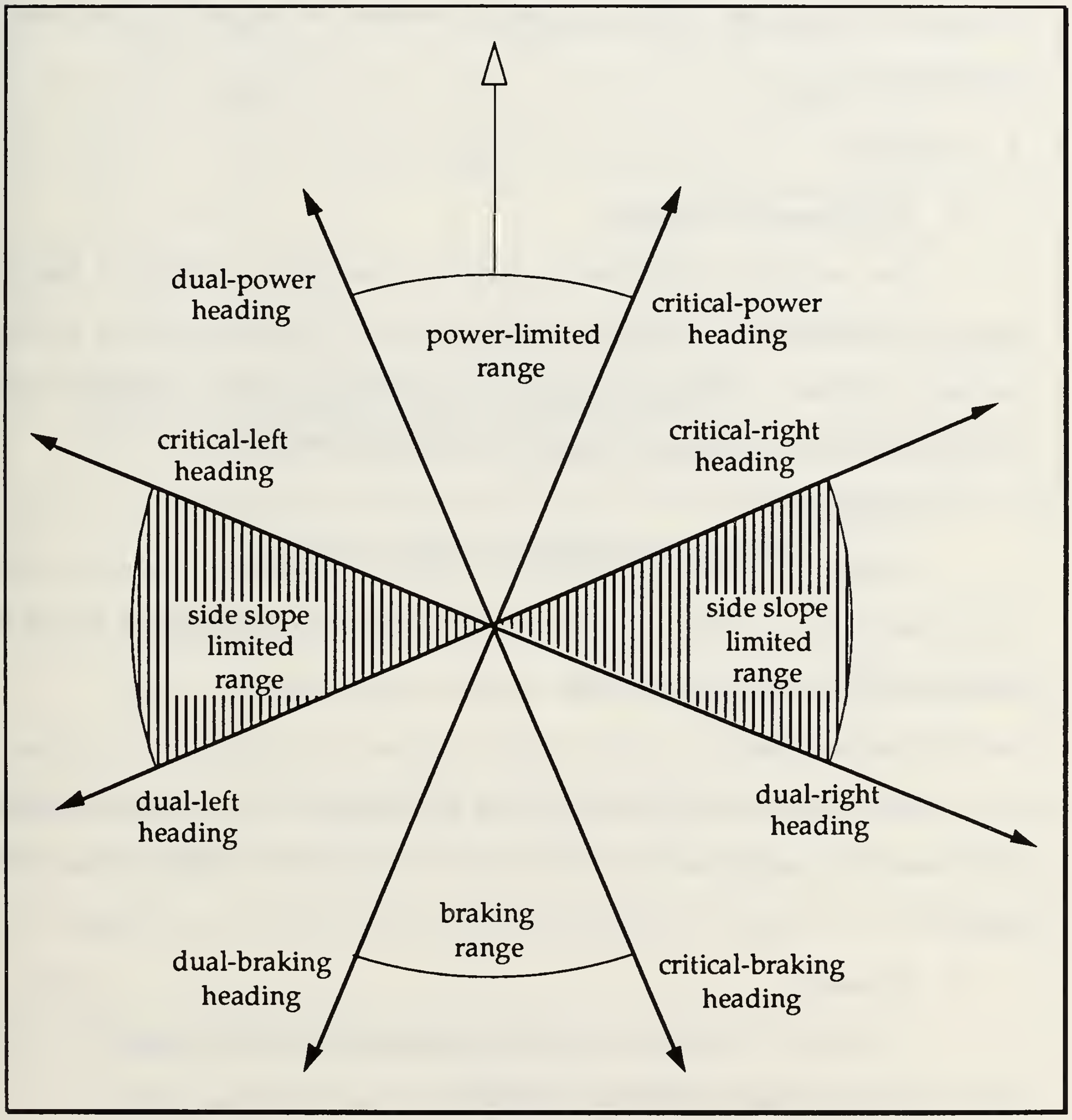


Figure 5. Critical heading ranges

These critical headings will vary from region to region as the slope varies. If a critical heading is equal to its dual, the region's slope is less than the vehicle's heading limit and no heading restriction exists for that particular range. For example, if a region's critical sideslope heading is equal to its dual sideslope heading, the vehicle can freely traverse the region without fear of rolling over.

B. TERRAIN

1. Homogeneous Regions

The terrain is modelled by *convex homogeneous polygonal regions*. Regions boundaries are defined by vertices and each region possesses uniform surface condition, slope and weight. The maps we used were manually constructed and do not represent actual terrain. See Figure 6.

2. Vertex List

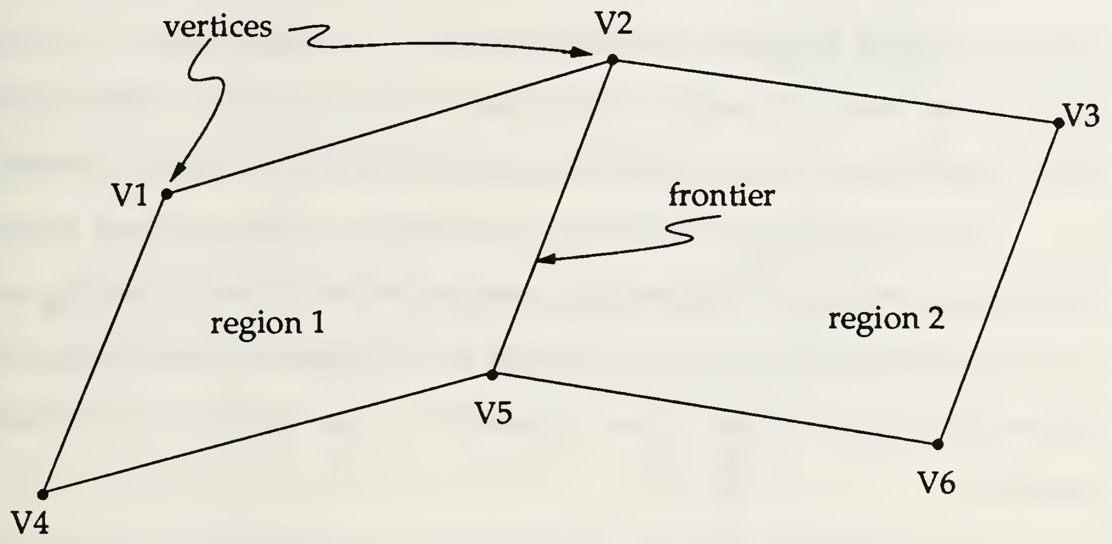
A *vertex* is a point defined by x, y, and z coordinates. A vertex can be on the boundary of more than one region. All vertices necessary to define the map are collected into a *vertex list*.

3. Region List

The *region list* is a collection of all regions in the map and contains the vertex list, weight, slope, orientation, and adjacent region list for each region.

4. Frontier

A *frontier* is the line separating two regions. It is represented as an edge with its endpoints defined by vertices.



Region 1 is defined by vertex list (V1, V2, V5, V6)
Region 2 is defined by vertex list (V2, V3, V6, V5)
The frontier between region 1 and region 2 is defined by edge (V5 V2)

Figure 6. Regions, Vertices, and Frontiers

4. Window list

Once an initial *window sequence* (a list of regions) is computed, each region in the list is put into the *window list* (a linked list). Additional information for the window list (such as all critical headings, and various "housekeeping" variables) is now calculated. We only compute these values for regions in the window list, and thus, only for the regions we will actually "visit. "

5. Type I, II, III, IV Traversals

Ross classified traversals through a region into four types:

A Type I traversal is the simplest. It is merely travel from one isotropic region to another isotropic region. In other words, the crossing point of the boundary edge is determined with respect to weighted distance only. There are no heading restrictions associated with this type of crossing.

With a Type II traversal the optimal path must be altered in one or both regions because it falls within a nonpermissible sideslope heading range. The nonpermissible heading is adjusted to the nearest critical angle, which allows the path to lie as close as possible to the optimal (but forbidden) heading.

Type III traversals feature switchbacks, which allow a vehicle to travel through an uphill nonpermissible heading range by switching between the pair of critical angles that border the nonpermissible range. The vehicle is allowed to make as many switchbacks as necessary to traverse the region.

Type IV involves travelling downslope where braking is required. There are no heading restrictions, but a different cost function (described in the next section) is required for travel within the braking range. See Figure 7.

7. Cost Functions

For Type I traversals, the cost function is simply the region's weight multiplied by the Euclidean distance calculated from the point of entry, *p1*, to the point of exit, *p2*.

$$c1 = \text{weight} \times \text{distance}\ (p1, p2)$$

For Type II traversals, the cost function is also weighted distance, as in Type I. However, recall that the heading is adjusted to a critical heading to allow passage through the region.

$$c2 = \text{weight} \times \text{distance}\ (p1, p2)$$

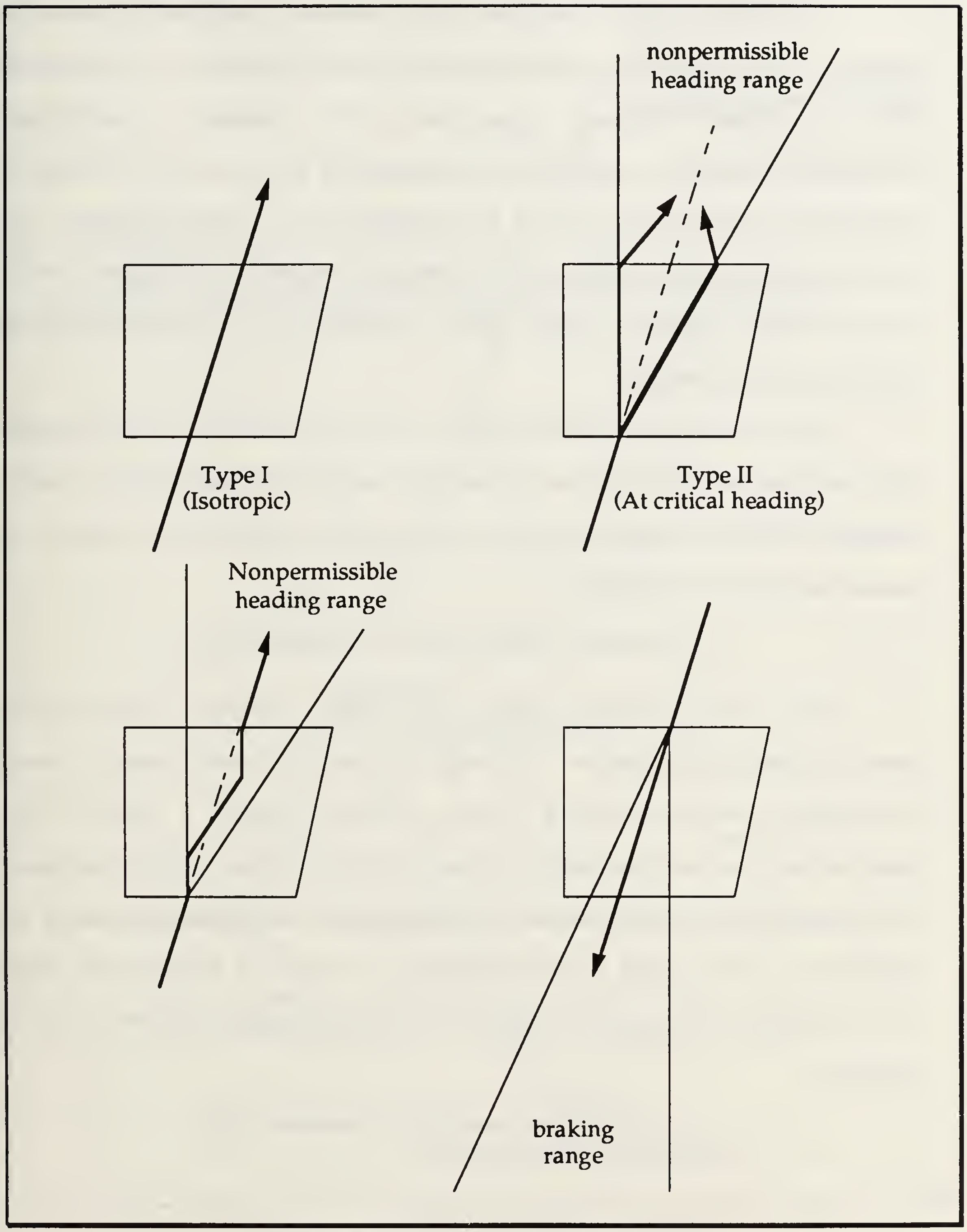


Figure 7. Type I, II, III, IV Traversals

Calculating Type III traversals also involves a weighted distance cost function, but the distance must take into account switchbacks. The distance for a switchback traversal, regardless of the number of switchbacks encountered, requires calculating the intersection of two rays. The first ray originates at start point *p1* of the first segment, *sw1*, with a direction of the critical impermissible heading. The second ray originates at the start point *p2* of the second segment, *sw2*, with a direction of π + the dual critical nonpermissible heading.

Call this point of intersection *i*. The cost is the weighted distance from the start point of the first segment to the point of intersection *i* plus the weighted distance from the point of intersection *i* to the start point of the second segment. See Figure 8.

$$c3 = \text{weight} \times \text{distance}\ (p1, i) + \text{distance}\ (i, p2)$$

The cost function for Type IV (braking) traversals must take into account changes in elevation. Although our vehicle model claims no energy is expended while travelling within a braking region, a virtual cost is incurred because potential energy is lost as a result of the vehicle's decrease in elevation [Ref. 6]. The cost for travelling within a braking region is thus calculated as the change in elevation from the point of entering the region, *p1*, to the point of exiting the region, *p2*. Distance is not relevant to this cost function.

$$c4 = \text{abs}\ ((\text{elevation}\ p1) - (\text{elevation}\ p2))$$

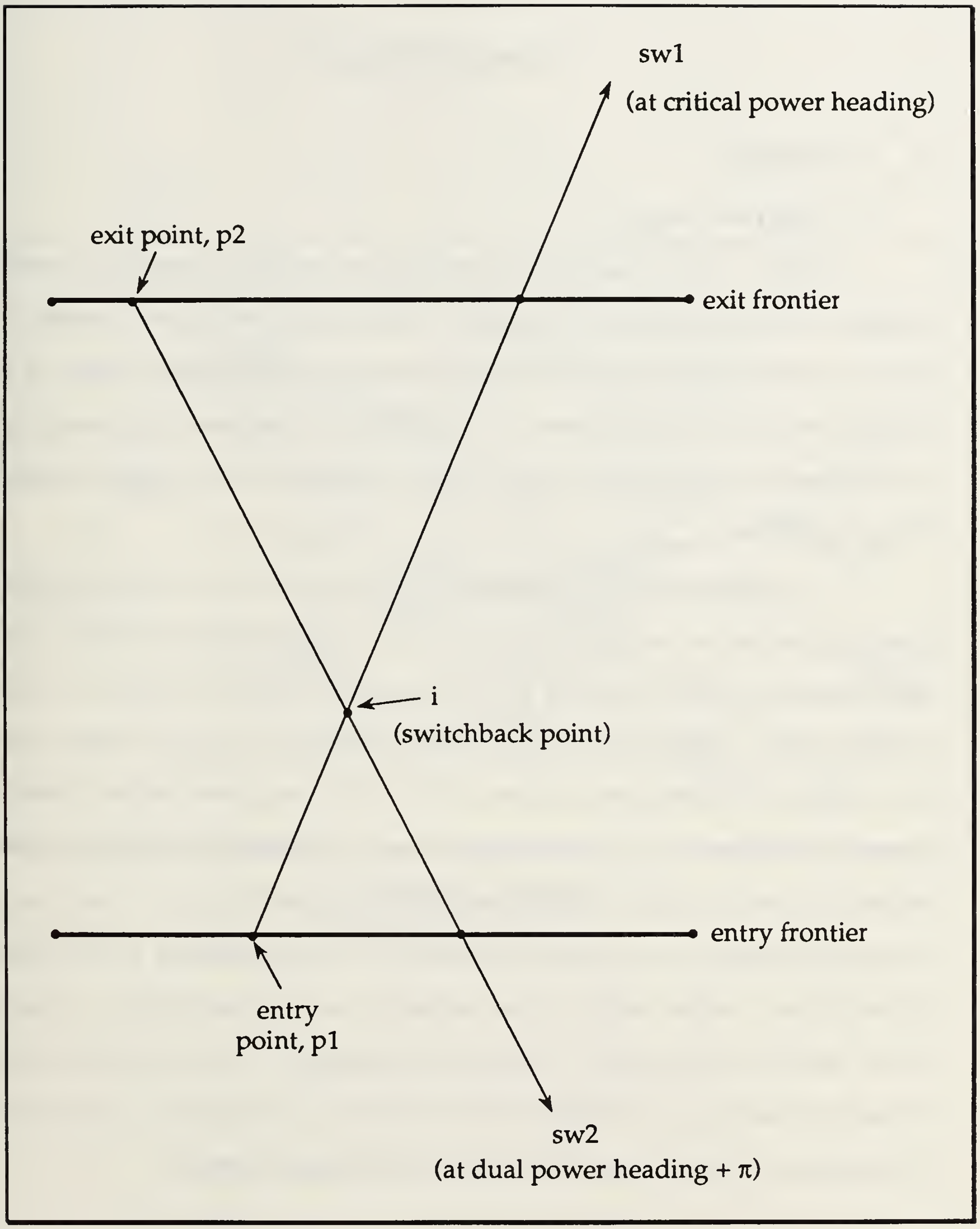


Figure 8. Finding a Switchback Point

V. PROBLEM SOLVING

A. A* SEARCH

1. Edge Dual Graph

Kindl's edge dual graph provides an excellent means of locating feasible window sequences. The fully connected edge mid-points (nodes) of a region provide a good cost estimate of traversing a particular portion of the region. See Figure 9. But instead of constructing a complete edge dual graph, we construct "on the fly." We only compute distances for the region currently being expanded.

At this point of the search we do not take into account vehicle heading restrictions (except as noted in the next paragraph), but we do expect every region in the sequence to be traversable within at least one more heading range. The A* search also avoids traversal of narrow regions where all headings between two frontiers fall within a non-permissible slideslope range. See Figure 10. Other regions may be labelled as obstacles if their traversal costs exceed a pre-determined threshold value. The window sequence returned by this search procedure is not guaranteed to be feasible. Although any single region in the sequence may be traversable, a crossing point between two regions which simultaneously avoids nonpermissible sideslope headings in both regions may not exist. See Figure 11. In this case, we attempt to detour around the impermissible path segment.

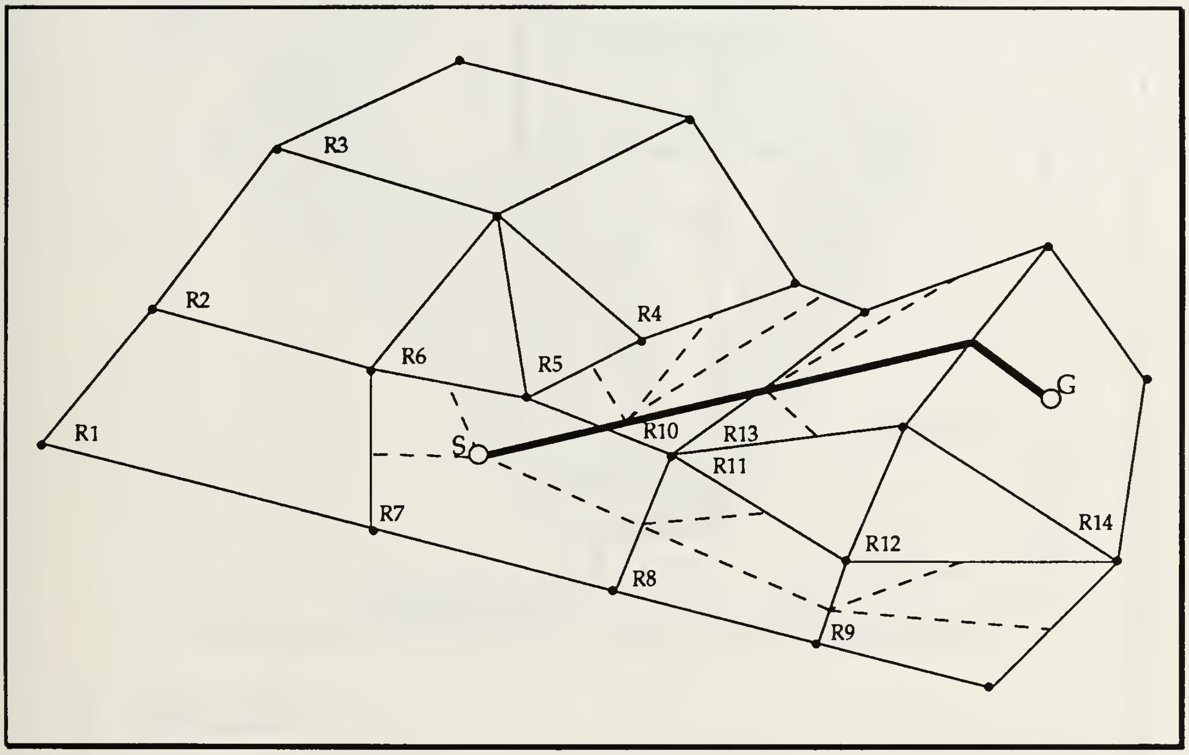
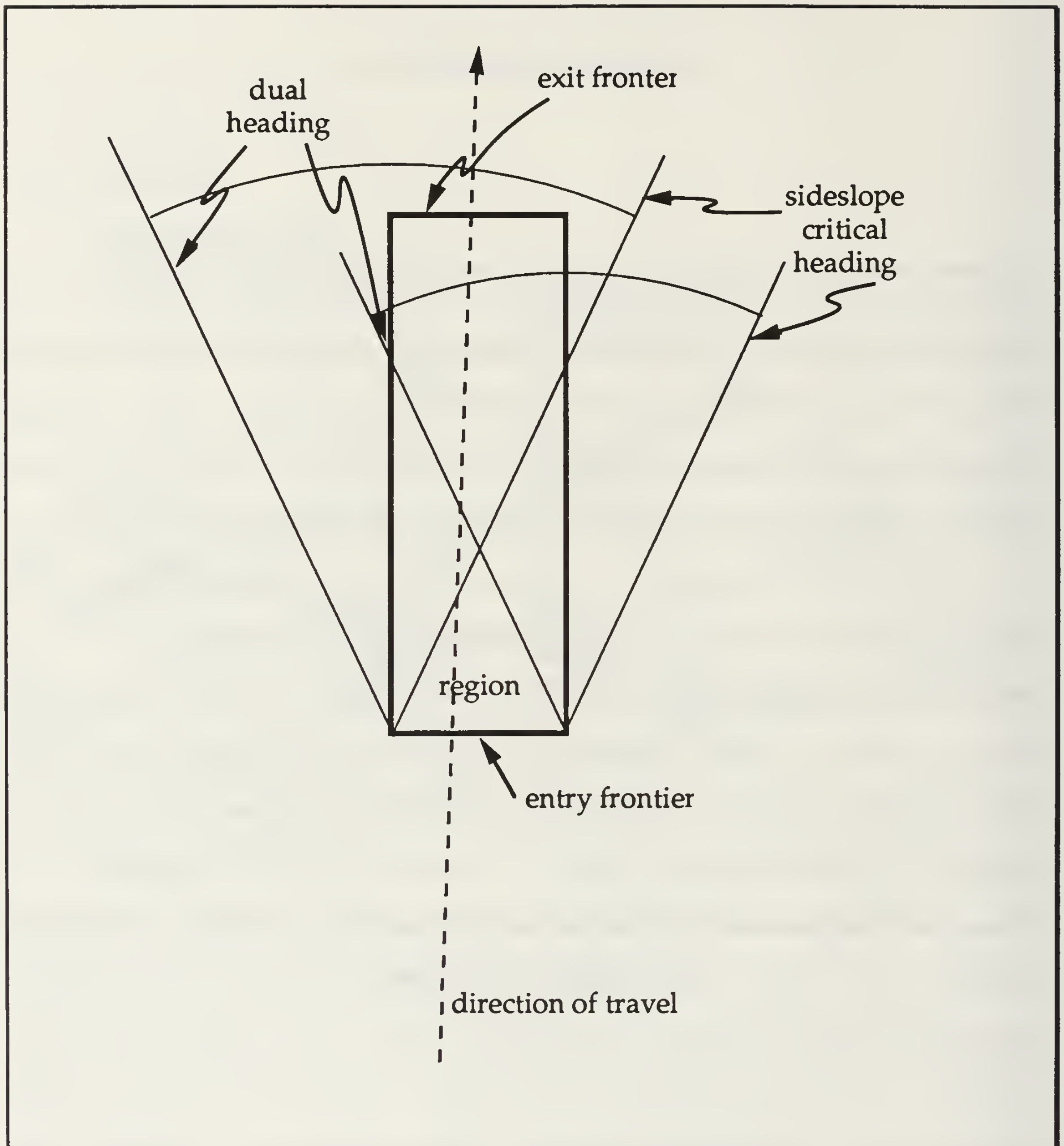


Figure 9. A* Search



It is impossible to travel from the entry frontier to the exit frontier without violating the sideslope heading restrictions.

Figure 10. Narrow Region with Nonpermissible Entry

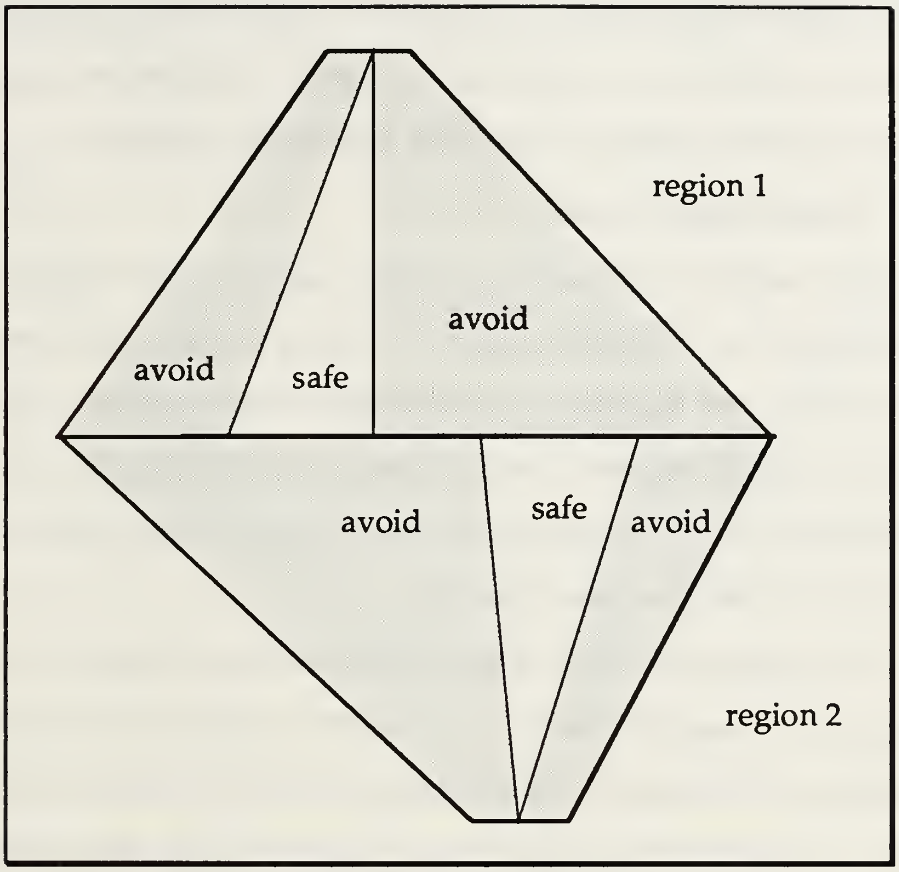


Figure 11. Two Regions with no Valid Crossing Point

2. Heuristic

The heuristic evaluation function for the A* search is the shortest distance to the goal. The distance from each active node (midpoint of the edge) to the destination is calculated and added to the *cumulative cost of* reaching the node. The node with the lowest sum is selected for expansion.

B. FINDING THE LOCALLY OPTIMAL PATH

1. Processing a Window Sequence

The A* search returns a list of regions within which a feasible path lies. The region list, start, and goal are processed into a linked list which contains region-specific information such as critical heading ranges, boundary edge and exit point, and a displacement which measures the magnitude of a point's movement during the iterative optimization process.

2. Golden Ratio Search

The golden ratio search is used to approximate the locally optimal path through two adjacent regions. The procedure involves two path segments, with one segment for each region being searched. The outer end points of the path segments remain fixed, and the interior point where the two segments meet is adjusted along the frontier (within a given tolerance) to find the minimum cost. See Figure 12.

The procedure exploits the convex function defined by adjusting the crossing point along the frontier. By sampling four points along the frontier, it is possible to determine that the optimal cost lies within the range of three of these points and the search space beyond these three points can be eliminated from the search process. This procedure can be repeated on an increasingly smaller search space until the desired accuracy is obtained. However, the procedure does reach a point of diminishing returns where additional searching produces negligible results. We halt the search when improvement is less than two percent of the current path cost

3. Partitioning the Regions by Heading Ranges.

When calculating the cost of traversing adjacent regions, we must determine within which heading range the vehicle is travelling.. The vehicle's heading determines which cost function to use. There are three permissible heading ranges: headings within the power range, braking range,

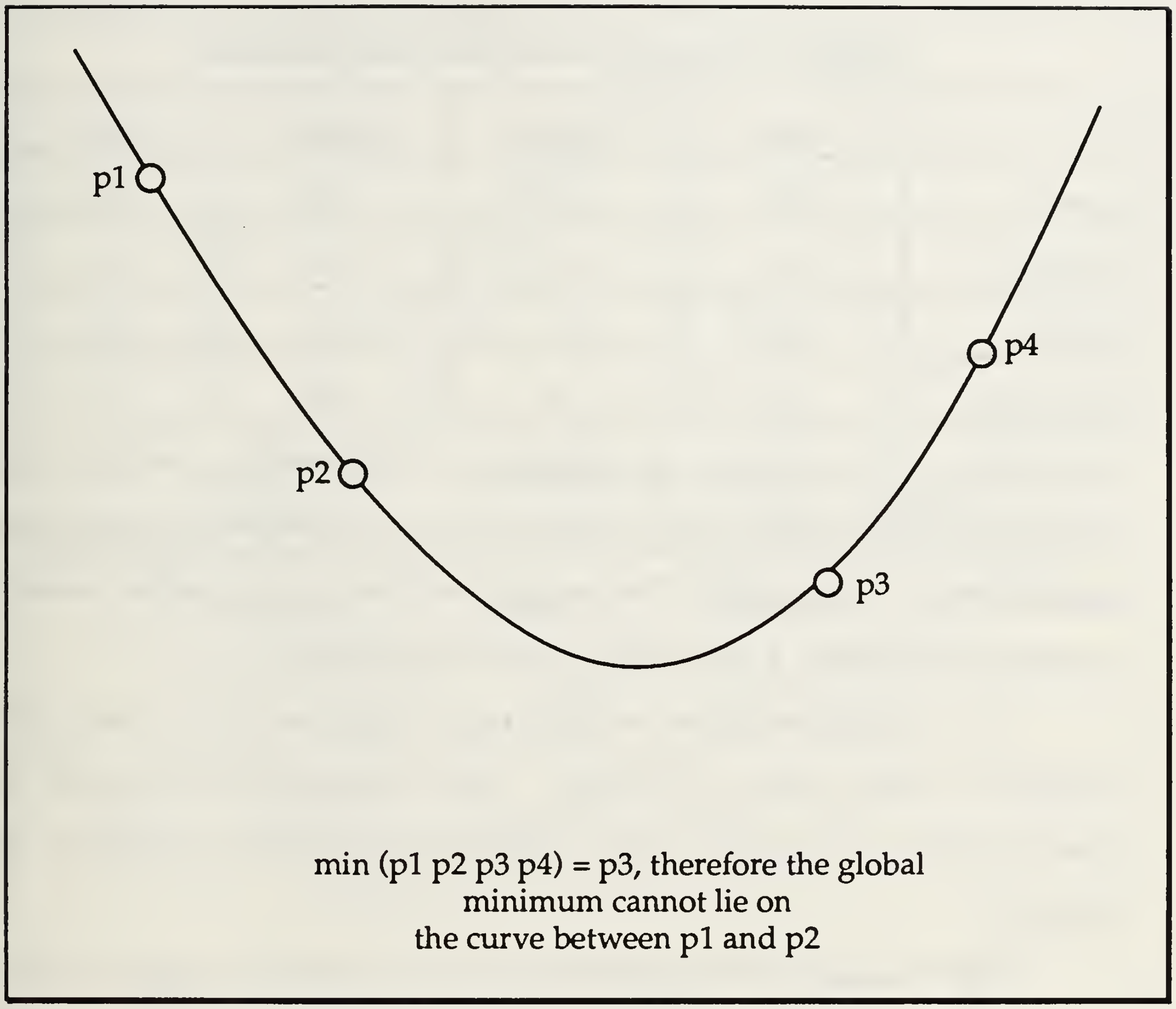

p1
p2
p3
p4
min (p1 p2 p3 p4) = p3, therefore the global
minimum cannot lie on
the curve between p1 and p2

Figure 12. Golden Ratio Search

and isotropic (unrestricted) range. We must also consider the nonpermissible sideslope heading ranges because an initial crossing point may force a path segment to lie within a nonpermissible heading range. Since the golden ratio search requires a convex cost function to approximate the cost of traversing two regions, we must consider the possible combinations of heading ranges (and thus cost functions) between two adjacent regions. There are $4^2 = 16$ possible combinations. See Table 1.

TABLE 1. HEADING RANGE COMBINATIONS

	power	isotropic	braking	side
power	pp	pi	pb	ps
isotropic	ip	ii	ib	is
braking	bp	bi	bb	bs
sideslope	sp	si	sb	ss

For example, "ip" represents the cost function for traversing a region within an isotropic heading range and transitioning to a power heading range when crossing into the next region. The ranges involving nonpermissible sideslope headings are assigned a large value representing "infinity."

We distinguish these heading range combinations by computing transition points along the frontier. Each transition point marks the transition from one heading range (and thus cost function) to another. See Figure 13.

The end points of the frontier serve as the initial and final transition points. We must also keep track of which region (the region before the frontier or after the frontier) and heading range the transition point is associated with. We now find intermediate transition points along the frontier, if any exist. For the first region, we compute a ray from the region's entry point at all eight critical headings. For every ray that intersects the frontier, we assign a transition point at the point of intersection. For the second region (beyond the frontier), we compute "back rays" from the region's exit point at each critical heading + π. We again assign a transition point where any back ray intersects the frontier.

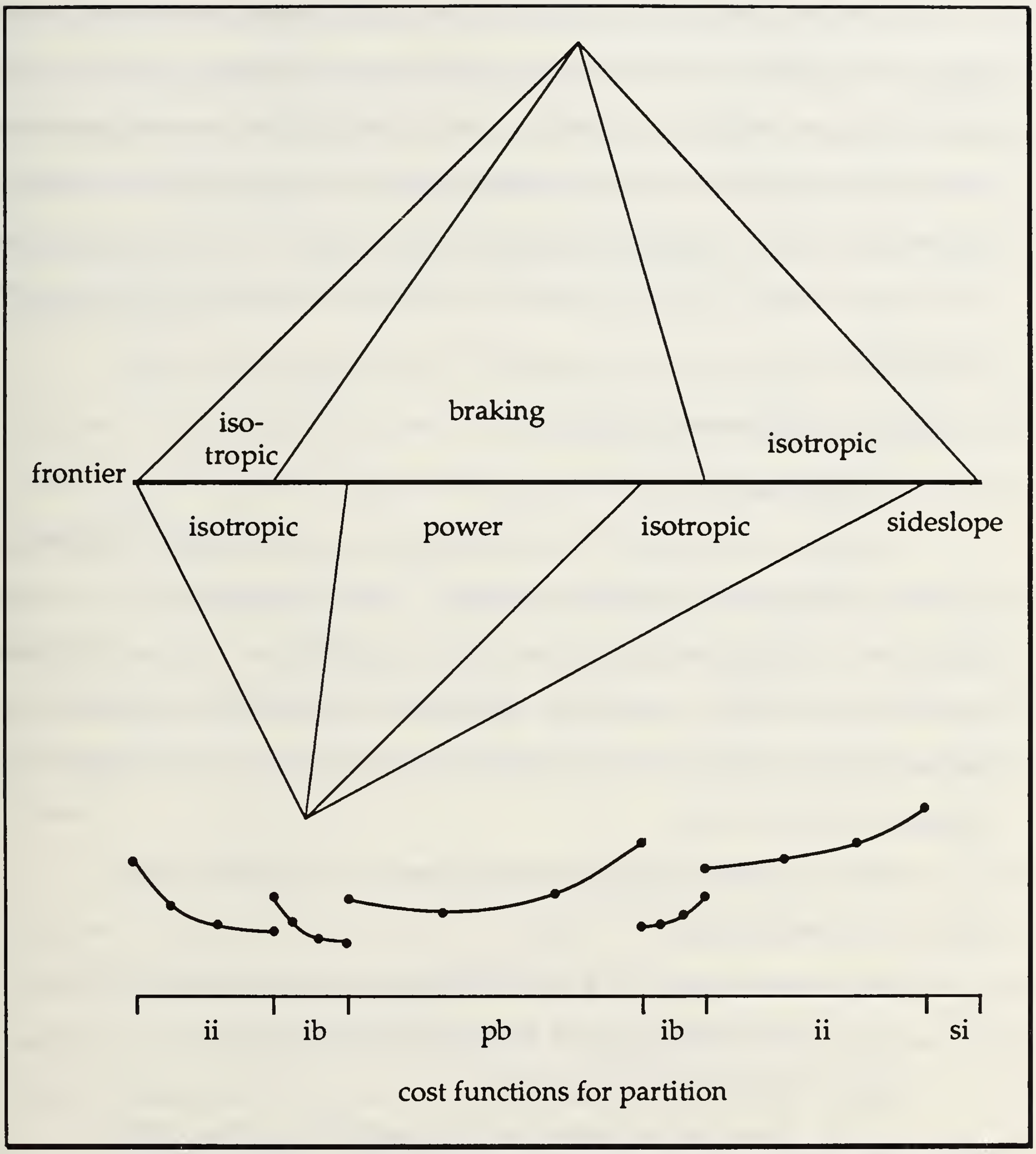


Figure 13. Partitioning Regions by Heading Ranges

Working from left to right along the frontier, we conduct a series of golden ratio searches. Each time we encounter a transition point, we change to the appropriate cost function.

After one sweep of the frontier, we have determined which heading range combination is likely to contain the optimal crossing point. Since we know between which two transition points this optimal crossing point lies (and thus the appropriate cost function to apply), we conduct a more precise golden ratio search along the frontier between these two transition points. This search locates (within tolerance) the optimal crossing point between the two regions.

C. SIMULATED ANNEALING

1. Background

Johnson, et al, [Ref. 7], describe the use of simulated annealing for solving the graph partitioning problem. This is essentially a modified technique of iterative improvement of a local solution until an acceptable result is achieved. Annealing allows for occasional acceptance of a degenerative solution in an attempt to escape from a locally optimal but globally poor solution.

The basic simulated annealing algorithm requires as input:

1) the solution space
2) control parameters for the annealing process, which include
   a) T_0 - the initial value of the control temperature T
   b) T_f - the "freezing" value of T
   c) R - the reduction factor for T (typically $0.70 <= R <= 0.99$)
   d) L - the maximum number of attempted moves at each value of T
   e) L_s - the maximum number of accepted moves at each value of T

Beginning with the initial solution, the move generator randomly perturbs this solution to obtain a new one. The change in cost, Δc, is the difference between the cost of the new solution and the current solution. If $\Delta c \leq 0$, the new cost is less than the current cost, and the new solution becomes the current solution. If $\Delta c > 0$, the new solution is accepted as the current solution based on a probability function. The typical probability function is $P = e^{\frac{-\Delta c}{T}}$, where Δc is the cost difference and T is the current temperature.

The control temperature T is expressed in the same units as the cost function. As the solution space is searched, T is reduced by the reduction factor R which progressively reduces the probability of accepting a solution with higher cost.

At each temperature T, the move generator attempts up to L moves, accepting up to L_s solutions of higher cost. The temperature eventually reduces to T_f, the freezing temperature, where no solutions with higher costs are accepted. At this point the algorithm halts and returns the best solution found.

Kindl [Refs. 4, 8] used this annealing concept in attempting to improve a current path solution. Recall that a path lies in a window sequence. The border between windows is an edge whose endpoints are defined by vertices. A vertex may define the endpoint of more than one edge. Thus, a vertex may be common to more than one window (or region).

Kindl's procedure randomly selects a vertex, called the rotation vertex, which belongs to the current window sequence. All edges with this vertex as an endpoint are identified. The current path solution passes through at least one of these edges. Edges through which the path crosses are removed from the window sequence and replaced by the edges through which the path does not pass. See Figure 14. This generates a path through the new window sequence which is iteratively improved (using the golden

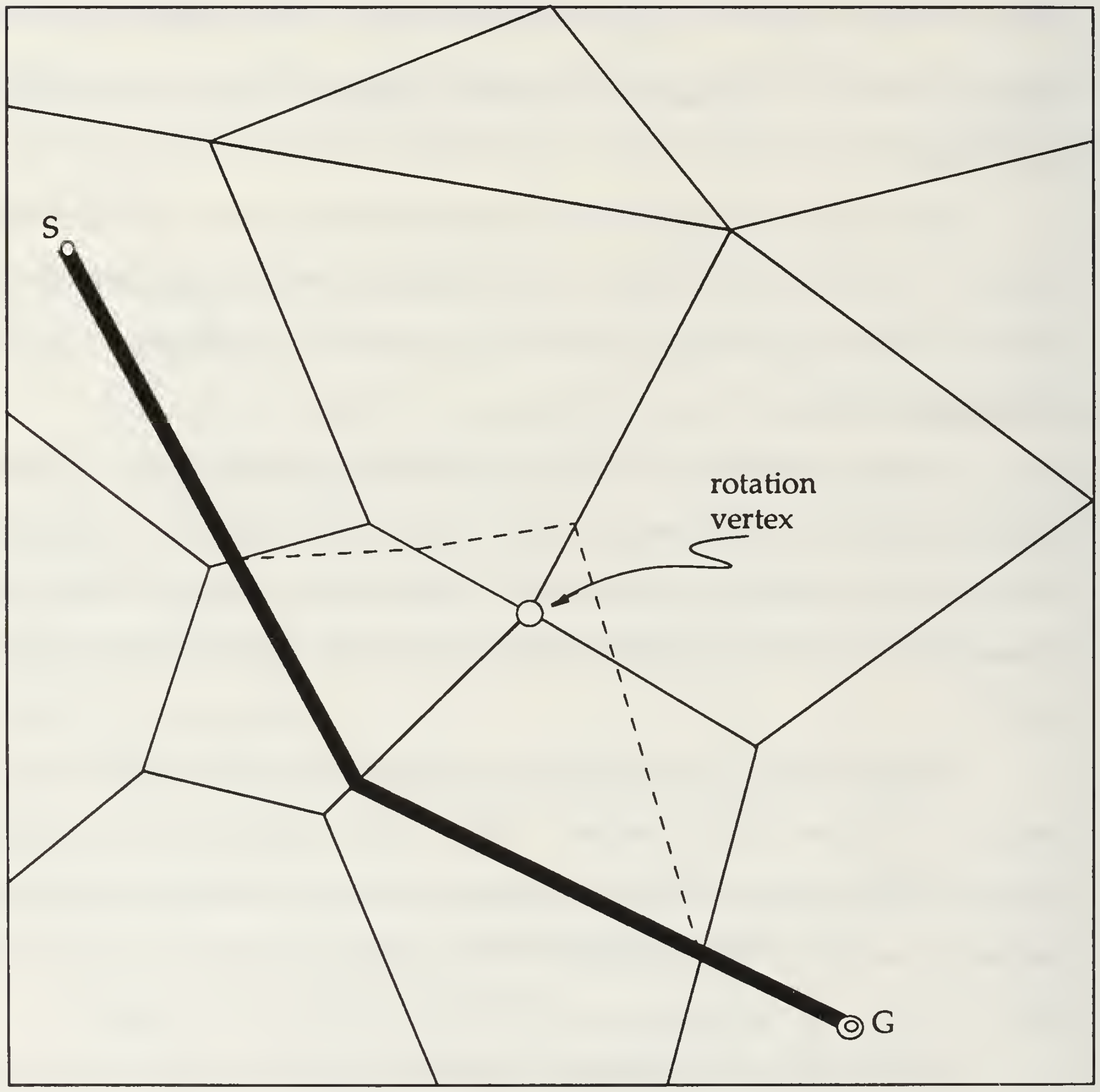


Figure 14. Rotation Vertex

ratio search) to achieve a locally optimal solution. If this path solution is better than the previous solution, the new path is accepted. If the new path solution is worse than the previous solution, the decision to accept it is determined by the probability function $P = e^{\frac{-\Delta c}{T}}$. This allows a means of escaping from a possible local minimum and searching other areas of the map for a better solution. See Figure 15. Kindl used the following control parameters:

T_0	= initial path cost
T_f	= 1.0
R	= .80
L	= 5
L_s	= 4

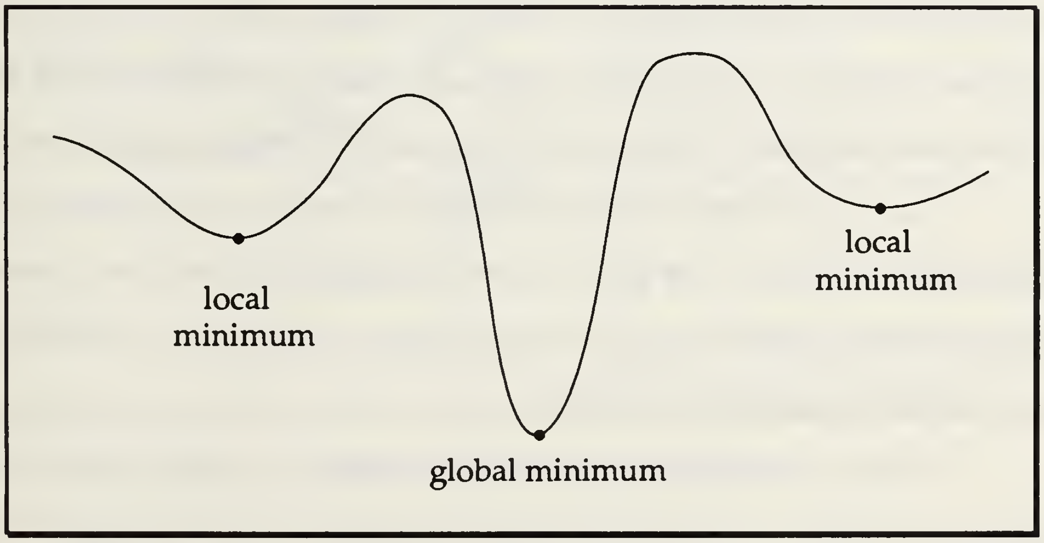


Figure 15. The Cost Function for Multiple Window Sequences

2. Path Annealing for Anisotropic Regions

The control mechanism Kindl used will also work for anisotropic path annealing. Temperature will control the program's probability of accepting worse or degenerative solutions.

The process which perturbs the path, or makes a "detour," is more complicated. A simple replacement of edges based on a rotation vertex may not produce a feasible window sequence. We must check the new edges to determine whether we can travel between them at a valid heading.

To do this, we select a *detour region*. We designate the entry point into the detour region as the *detour start point*. We assign the exit frontier of the detour region as the *detour frontier*. The exit point of the region immediately beyond the detour region is assigned as the *detour goal point*. We now conduct an A* search similar to the search performed to produce the initial window list. The search begins at the detour start point and terminates under one of two conditions: 1) the detour goal point is reached, or 2) an *upstream region* is reached. An upstream region is a region in the window list between the detour region and the (global) goal region. See Figure 16.

For the A* search we use frontier endpoints in addition to frontier midpoints as nodes. This gives us greater resolution and often results in a shorter detour. This procedure is particularly useful when searching across regions that vary greatly in width.

This A* search is more restrictive than the search used to find the initial region list. In addition to checking for valid heading ranges between regions, we limit the length of the detour to no more than three regions. This "tames" the search and prevents it from producing a circuitous detour which would likely result in a path solution considerably longer than the current solution.

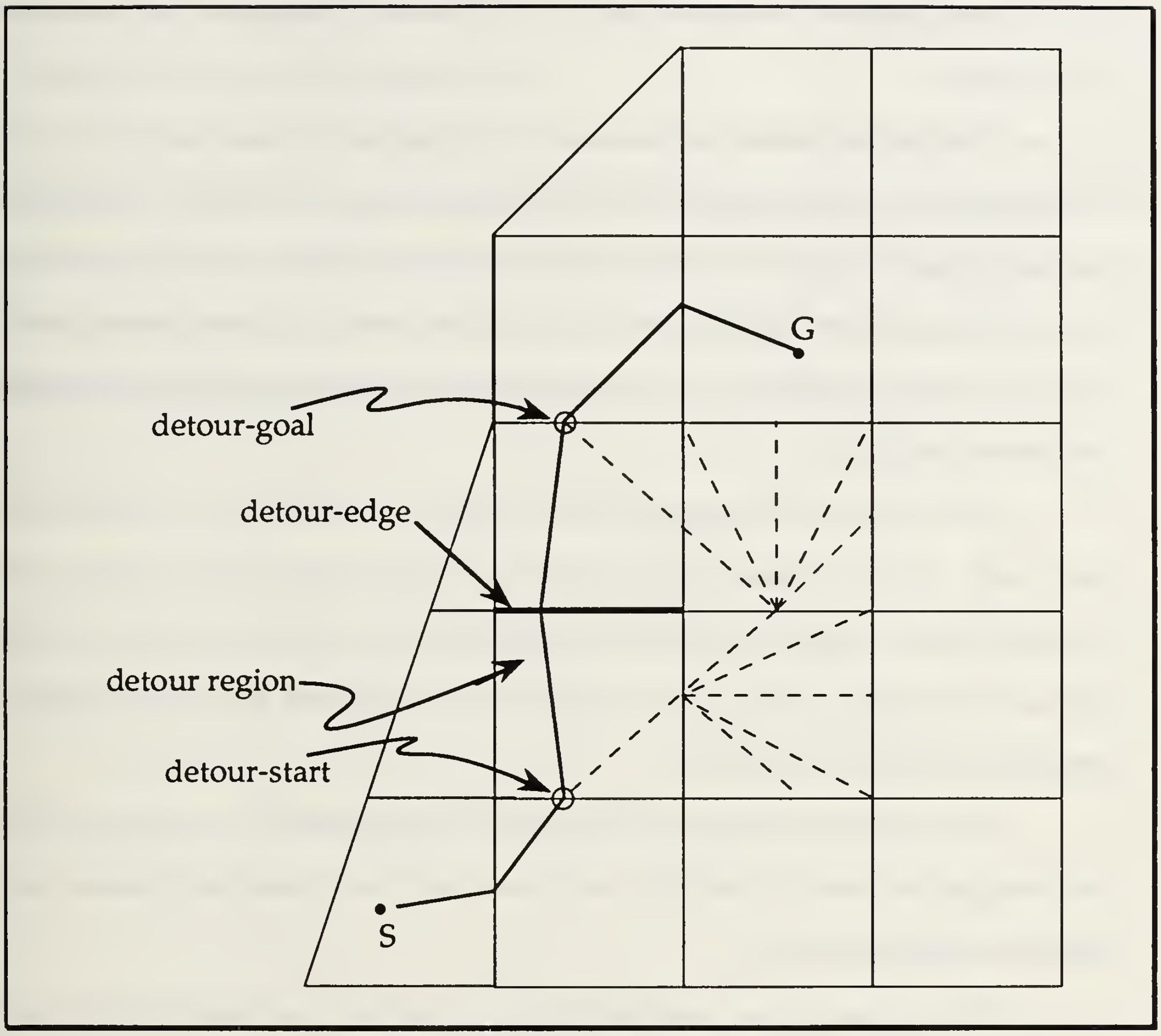


Figure 16. Finding a Detour

This procedure effectively eliminates many regions as candidate detour regions. This is desirable, since we don't want to waste our time on detours unlikely to produce a better path solution.

The A* search produces a *detour region list*. The detour region list is processed into a window list and inserted into the current window list, replacing the detour region(s). The resulting window list will contain at least one different region where the detour occurs, but it may contain up to two new regions.

The golden ratio search is iteratively applied to the new window list to determine a locally optimal (within tolerance) path solution. The frontier crossing points of the windows retained from the previous window list serve as a starting solution for this next search. The result is a faster search, since a locally optimal solution already exists for path segments before and beyond the detour region(s).

Once the locally optimal path is found, we compare it to the current best path. If the current path is shorter, it becomes the best path. If the current path is longer than the best path and not accepted by the annealing control mechanism, the previous window list is restored and a new region is randomly selected for a detour.

The number of detours attempted is determined by a loop structure and the preset value L, and the maximum number of accepted moves any a given temperature is L_s.

We initially use the probability function mentioned previously, $P = e^{\frac{-\Delta C}{T}}$. We use Kindl's values for control parameters, except for R, where we use .70.

Since our relatively simple map provides few opportunities to exploit path annealing, we also experiment with a linear probability function $P = T$. The number of detour attempts allowed is a function of the length of the window list. We also discard the value L_s, which governs the maximum number of accepted moves at a given value of T. The temperature T is reduced by a constant R (i.e. T = T – R) after every L detour attempts. The results are compared in Chapter VI.

We always keep track of the current path solution and the best path solution. We also maintain a path list. A new path solution is added to the path list if it is accepted as the current solution. The path list allows us to provide alternate path solutions which may be longer than the best path, but are nevertheless feasible path solutions.

VI. EMPIRICAL STUDIES

A. INPUT MAPS AND VEHICLES

1. Maps

Our input map, derived from Ross, is not particularly suited to exploiting the annealing process. The regions are largely symmetrical and oblong. This arrangement often permits the A* search to produce the optimal window list instead of relying on the annealing process to find it. In this case, most of the effort is spent in merely confirming that we have already located the best path. However, there are some cases where annealing produces a better solution. In our test cases, we display the path solution produced by the A* search as well as the solution found by annealing.

We use two maps. The maps appear identical, but the second map contains regions weighted in proportion to their slopes. All regions in the first map have a weight of 1. Figures 17 and 18 are included to show map features. Figure 17 shows a path found by the A* search. Figure 18 shows the path solution after annealing. Heading range indicators for the vehicle are drawn at each point where the path crosses a frontier.

2. Vehicle

We use three vehicles for the evaluation. Vehicle-1 and vehicle-3 are adapted from Ross [Ref. 1] and represent, respectively, an armored personnel carrier and a cargo truck. Vehicle-2 cannot climb as steep a slope as vehicle-1, but its sideslope stability is greater. We did this to see how much the path between a common start and goal varies for different vehicles.

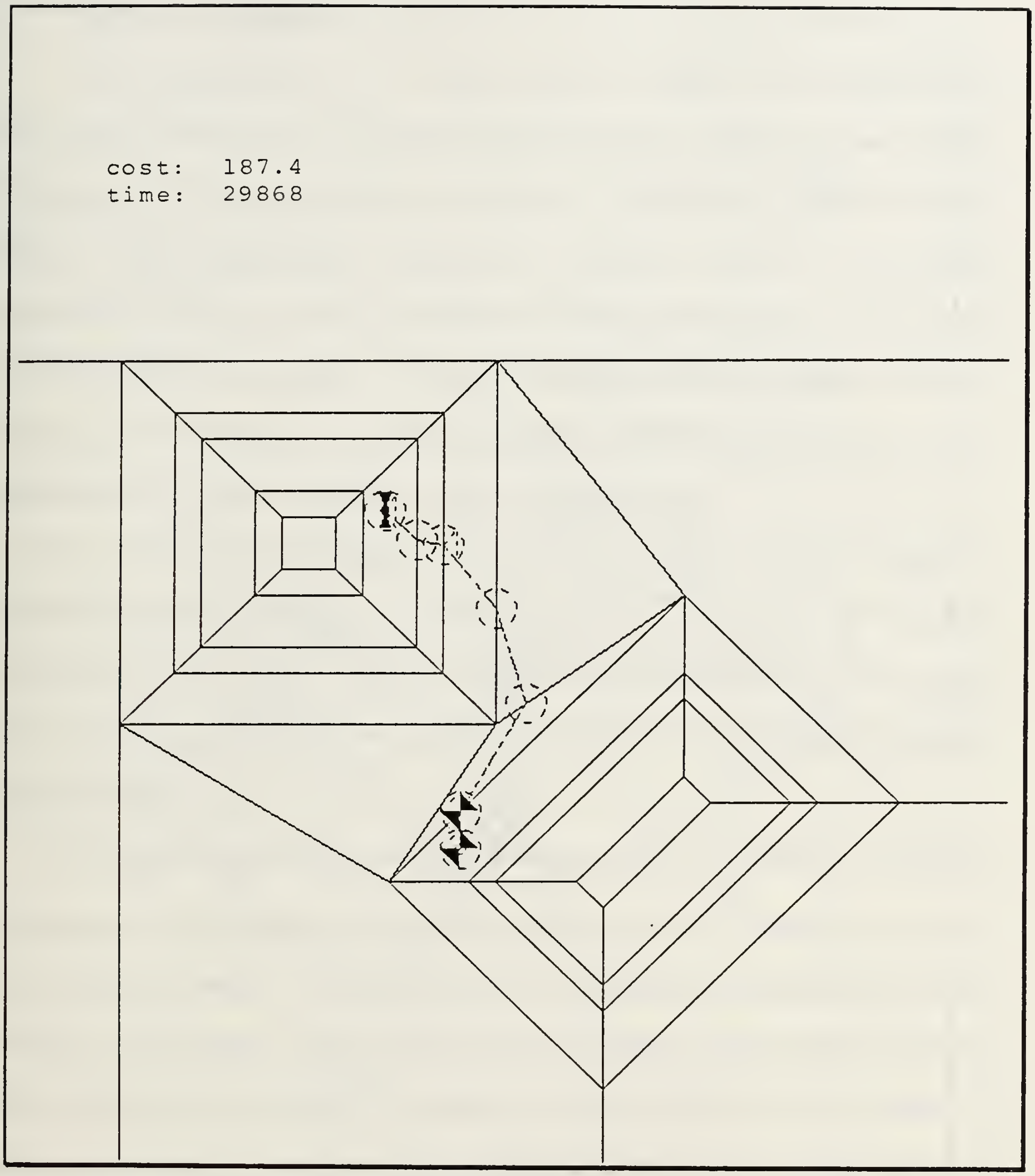


Figure 17. Path before Annealing

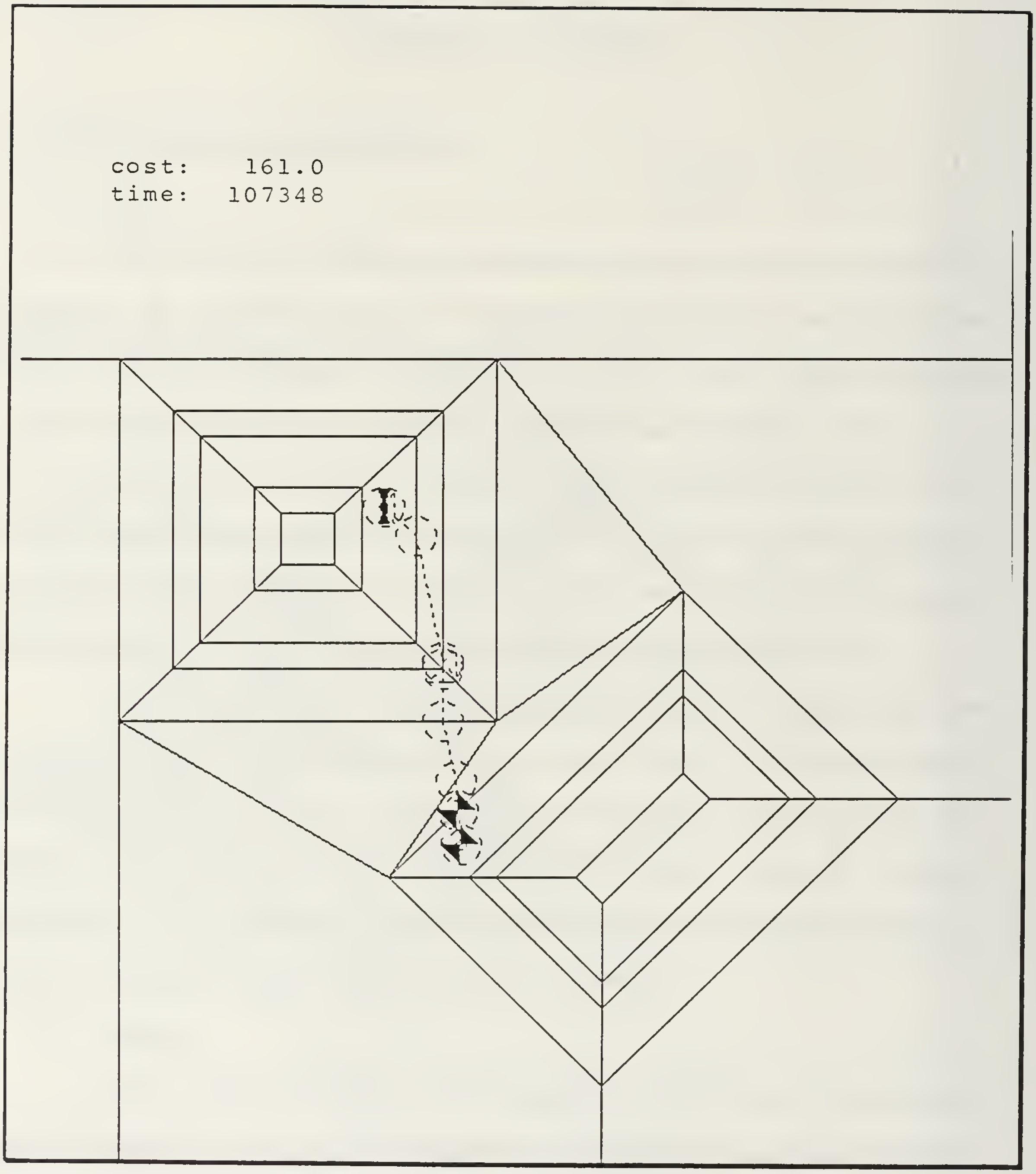


Figure 18. Path after Annealing

B. TEST PROCEDURES

A direct comparison of our implementation to Ross's is not possible since Ross used Flavors on a Symbolics LISP machine while we use Allegro Common LISP on a Solbourne workstation. We are not certain whether Ross's time computations include all overhead, output, and display procedures. Also, our map is slightly modified from Ross's. The slope of some regions was changed to be more consistent with vertex elevations. All regions in Ross's map are equally weighted, therefore a comparison of our performance with the weighted map would be invalid.

We do observe the performance of our implementation based on different vehicles, temperature cooling schedules, and maps. For our time computation (msec), we include all processes, beginning with input of the start and goal points. We also include time allocated to updating displays and input/output.

We evaluate the path annealing process with two different probability functions, the cost-based exponential function $P = e^{\frac{-\Delta C}{T}}$, and the linearly decreasing function P = T. Table 2 shows the results of this comparison for vehicle-1 and Table 3 shows the results for vehicle-2. The temperature and the initial path cost in the exponential function influences the number of detours attempted as well as the probability of accepting a degenerative path solution. In the linearly decreasing function, the number of detours attempted is dependent on length of the window list and the probability of accepting a degenerative path solution begins with a constant initial temperature and decreases at a constant rate during each iteration until it reaches the freezing temperature.

TABLE 2. EQUALLY WEIGHTED REGIONS WITH VEHICLE-1

vehicle-1		A*		linear p		exponential p	
start	goal	cost	time	cost	time	cost	time
(221.0 399.0)	(260.0 193.0)	187.4	29868	161.0	107348	187.4	159951
(229.0 127.0)	(305.0 344.0)	233.2	16850	204.8	34284	204.7	40784
(399.0 217.0)	(561.0 239.0)	127.4	15467	127.4	41599	127.4	64667
(271.0 415.0)	(266.0 194.0)	208.8	18317	199.1	58517	199.1	87466
(77.0 339.0)	(364.0 297.0)	222.8	23917	222.8	100617	222.8	144850

TABLE 3. EQUALLY WEIGHTED REGIONS WITH VEHICLE-2

vehicle-2		A*		linear p		exponential p	
start	goal	cost	time	cost	time	cost	time
(221.0 399.0)	(260.0 193.0)	127.7	35483	109.7	82200	109.7	217533
(229.0 127.0)	(305.0 344.0)	156.3	20600	156.3	39366	156.3	115684
(399.0 217.0)	(561.0 239.0)	89.0	17434	51.15	103717	80.5	107083
(271.0 415.0)	(266.0 194.0)	176.7	21366	176.7	39700	176.7	88450
(77.0 339.0)	(364.0 297.0)	222.8	24199	217.9	63501	222.8	179266

We also tested the annealing process on the weighted region map. The results are shown in Tables 4 and 5.

TABLE 4. WEIGHT PROPORTIONAL TO SLOPES WITH VEHICLE-1

vehicle-1		A*		linear p		exponential p	
start	goal	cost	time	cost	time	cost	time
(221.0 399.0)	(260.0 193.0)	282.4	27550	259.1	16517	253.1	201034
(229.0 127.0)	(305.0 344.0)	238.6	10516	238.6	16001	238.6	30100
(399.0 217.0)	(561.0 239.0)	109.7	31466	109.7	14900	109.7	52066
(271.0 415.0)	(266.0 194.0)	328.8	17834	328.8	38183	328.8	69000
(77.0 339.0)	(364.0 297.0)	433.4	20889	433.4	96483	433.4	46949

TABLE 5. WEIGHT PROPORTIONAL TO SLOPE WITH VEHICLE-2

vehicle-2		A*		linear p		exponential p	
start	goal	cost	time	cost	time	cost	time
(221.0 399.0)	(260.0 193.0)	281.1	30449	216.0	158816	278.4	229483
(229.0 127.0)	(305.0 344.0)	238.6	10533	238.6	35384	238.6	62167
(399.0 217.0)	(561.0 239.0)	79.1	38233	79.1	44634	79.1	90000
(271.0 415.0)	(266.0 194.0)	347.8	20966	347.8	29784	347.8	68850
(77.0 339.0)	(364.0 297.0)	445.6	20617	445.6	46234	445.6	96284

C. ANALYSIS OF TEST RESULTS.

The results support the conclusion that path annealing can improve upon an initial A* search solution in the $2\frac{1}{2}$ dimensional weighted region problem. However, the stochastic approach may possess disadvantages. Because detours are chosen randomly, we are not guaranteed of achieving the same solution for multiple runs of the program. Even if multiple runs produce the same solution, we don't know *when* the best solution was found. It may be found on the first detour, or it may be found only after making many detours.

If time is limited, we should be able to halt the program prematurely and return the current best solution. The randomness of choosing a detour region virtually assures that a program repeatedly halted at the same point will return different current solutions for the same start and goal.

A possible cure for this is to select a detour region based on a heuristic evaluation. We should attempt to detour through regions with a high potential for reducing the current path solution. A relatively high cost region or a path segment with a sharp change in direction may be promising candidates for a detour. This area warrants further study.

In our test cases, the linearly decreasing probability function works as well as the exponential probability function as far as finding the best path. Execution time is less for the former, however, this observation is based on very limited testing and is probably dependent on the shapes and sizes of map regions.

VII. SUMMARY AND CONCLUSIONS

Finding the optimal path solution through complex and varied terrain is expensive in terms of time and computer resources. Often heuristics can be applied to find a near-optimal or otherwise acceptable path solution in much less time. Several efforts have been made to locate feasible paths through two-dimensional terrain. This work expands Kindl's stochastic path planning approach to the $2\frac{1}{2}$ dimensional weighted region problem.

It is important to begin with a good initial window list. This is accomplished by an A* search. The effectiveness of path annealing is probably dependent on characteristics of the terrain, but further study is needed in this area. Random annealing works, but a heuristic approach to annealing should be explored. For example, it may prove beneficial to detour around high cost regions.

To be certain of finding the globally optimal solution, one must exhaustively search every feasible window sequence. Ross shows that this can be prohibitively expensive in terms of time and computational effort. A heuristic and stochastic approach produces a timely and acceptable path solution, but random detours will not always return the same path solution for a given start and goal.

The basic concept is shown to work in the LISP implementation. We do not guarantee an optimal or near-optimal path solution will be found. We do begin with a feasible starting solution and iteratiavely attempt to improve it under the control of the golden ratio search tolerances and the annealing temperature.

Suggested extensions to this research include testing and evaluating the performance of this implementation on various terrain models, incorporating a terrain mapping program, developing heuristic evaluation functions for selecting detour regions, and the use of parallel processing to simultaneously compute multiple window sequences.

LIST OF REFERENCES

1. Richbourg, R. F. , *Solving a Class of Spatial Reasoning Problems: Minimal-Cost Path Planning in the Cartesian Plane,* Ph. D. Dissertation, Computer Science Department, Naval Postgraduate School, Monterey, California, June 1987.

2. Mitchell, J. S. B. and Papadimtriiou, C. H. , "The Weighted Region Problem: Finding Shortest Paths Through a Weighted Planar Subdivision," *Journal of the ACM,* 1990.

3. Ross, R. S. , *Planning Minimum-Energy Paths in a Off-Road Environment with Anisotropic Traversal Costs and Motion Constraints,* Ph. D. Dissertation, Naval Postgraduate School, Monterey, California, June 1989.

4. Kindl, M. R., *A Stochastic Approach to Path Planning in the Weighted Region Problem,* Ph. D. Dissertation, Naval Postgraduate School, Monterey, California, March 1991.

5. Ross, R. S. and Rowe, N. C. , *Optimal Grid-Free Path Planning Across Arbitrarily Contoured Terrain with Anisotropic Friction and Gravity Effects,* Naval Postgraduate School, Monterey, California.

6. Rowe, N. C. and Kanayama, Y. , *Minimum-Energy Paths on a Vertical-Axis Cone with Anisotropic Friction and Gravity Effects,* Naval Postgraduate School, Monterey, California.

7. Johnson, D. S. , Aragon, C. R. , McGeoch, L. A. and Schevon, C. , *Optimization by Simulated Annealing: An Experimental Evaluation; Part I, Graph Partitioning," Operations Research,* v. 37, no. 6, November-December 1989.

8. Kindl, M. R. , Rowe, N. C. and Shing, M. , *Solving the Weighted Region Problem with Simulated Annealing,* Naval Postgraduate School, Monterey, California.

APPENDIX A. NOTES ON THE IMPLEMENTATION

1. The implementation program was written in Common LISP with X Windows and run on a Solburne Workstation. The system possesses its own operating system, 16 Megabytes of RAM, and runs at a clock speed of 33 MHz.

2. The emphasis was on producing a working prototype with efficiency a secondary priority. Many of the routines used in this program were more efficiently implemented using C++ and Prolog by Kindl. We perform very little pre-processing on the map data, preferring instead to calculate as we conduct the search. In an application where one map is extensively used, pre-processing would undoubtedly prove more efficient.

3. We used only the simplest data structures. The program is essentially constructed with LISP's defstruct, cond, if, dolist, cons, append functions. Sequential doubly linked lists were used instead of arrays, hash tables, or binary trees. The intent was to produce simple and readable code (as far as this is possible in LISP).

4. Local variables were primarily used. We only used global variables for such data as the start/goal points, window lists, region lists, accuracy control variables, and annealing control variables.

5. There is no elaborate user interface. The user can use the setf command to change vehicles. The start and goal points are input by using the mouse.

APPENDIX B . SOURCE CODE (COMMON LISP)

```
;*********************************************************************
;
;  Filename:  start.cl
;
;  By:  HILTON, Cary A.
;
;  Date:  May 91
;
;  Function:  Startup file for anisotropic path planning.
;
;             1.  creates and activates window *map*
;
;             2.  loads file "find.cl"
;
;             3.  to begin, type "(path)", then
;                 click left mouse button on
;                 start, goal points
;
;*********************************************************************


(use-package  :cw)
(require  :xcw)
(initialize-common-windows)

(defvar *map*)
(defvar *first-win*)
(defvar *win-num*)

(defvar *x-min* 0)
;(defvar *x-max* 695)
(defvar *x-max* 600)
(defvar *y-min* 25)
;(defvar *y-max* 838)
(defvar *y-max* 830)


(setf *map*
  (make-window-stream
   :left    *x-min*
   :bottom  *y-min*
   :width   *x-max*
   :height  *y-max*
   :title   "MAP WINDOW"))

(activate *map*)

(load "find.cl")
```

```
;*******************************************************************
;
;  Filename:  find.cl
;
;  By:  HILTON, Cary A.
;
;  Date:  May 91
;
;  Function:  Control structures for program.
;             1.  loads all files necessary
;                 to run program (except "start.cl")
;             2.  sets default maps and vehicles
;             3.  contains top level control program
;
;******************************************************************


(load "anneal.cl")
(load "converge.cl")
(load "costf.cl")
(load "drawf.cl")
(load "geometry.cl")
(load "links.cl")
(load "partition.cl")
(load "search.cl")
(load "vehicle.cl")
(load "map2.cl")

;  commands to load compiled files

;(load "anneal.fasl")
;(load "converge.fasl")
;(load "costf.fasl")
;(load "drawf.fasl")
;(load "geometry.fasl")
;(load "links.fasl")
;(load "partition.fasl")
;(load "search.fasl")
;(load "vehicle.fasl")

;(load "map2.fasl")
;*****************************************************

(defstruct path
  region
  point
  heading
  cost)

(defvar *start*)       ; global start point
(defvar *goal*)        ; global goal-point
(defvar *rlist*)       ; path passes through region list
(defvar *winlist*)     ; processed window list
(defvar *init-win*)    ; first window in window list
```

```
(defvar *first-win*)   ; second window in window list

(defvar *path-list*    nil)
(defvar *current-path* nil)
(defvar *best-path*    nil)
(defvar *new-path* nil)
(defvar *random-value*)
(defvar *accept-path*  .5)   ; this is the annealing "temperature"
                             ;  when temperature reaches 0, no
                             ;  degenerate paths will be accepted

(defvar *l-move* 4)
(defvar *l-accept* 4)
(defvar *reduction-factor* 0.7)
(defvar *temp-f* 1.0)
(defconstant *e* 2.71828)

;*****************************************************

;  Default vehicle is vehicle-1

(setf *vehicle* (vehicle-rad vehicle-1))

;(setf *vehicle* (vehicle-rad vehicle-2))

;(setf *vehicle* (vehicle-rad vehicle-3))

;(load "map3.cl")

;(load "map3.fasl")

;*****************************************************

(defun get-start ()
  (let (mouse-p)
    (pprint "click mouse on start point")
    (setf mouse-p (get-position *map*))
    (setf *start* (cons (float (position-x mouse-p))
                 (list  (float (position-y mouse-p)))))
    (draw-start *start*)
    (pprint *start*)))

(defun get-goal ()
  (let (mouse-p)
    (pprint "click mouse on goal point")
    (setf mouse-p (get-position *map*))
    (setf *goal* (cons (float (position-x mouse-p))
                 (list (float (position-y mouse-p)))))
    (draw-goal *goal*)
    (pprint *goal*)))

(defun create-window-list (start goal)
  (setf *winlist* (change-struct start goal *rlist*))
  (double-link (first *winlist*) (rest *winlist*))
  (setf *init-win*  (first *winlist*))
  (setf *first-win* (segment-next-win *init-win*))
  (setf *win-num* (segment-index (first (last *winlist*)))))

(defun path ()
  (setf *best-path* nil)
  (setf *path-list* nil)
  (setf *current-path* nil)
  (clear *map*)
  (draw-regions *global-region-list*)
;  (get-start)
;  (get-goal)
  (find-path *start* *goal*)
  (loop
    (cond (*bad-segments*
           (detour-at (first *bad-segments*))
           (qdraw)
         (converge *first-win*)))
;   (draw-final *first-win*)
    (if (null *bad-segments*)
      (return)))
    (setf *current-path* (store-path *first-win*))
    (setf *best-path* *current-path*)
    (setf *path-list* (append *path-list*
                             (list *current-path*)))
;    (anneal-path)  ; linear cooling
;    (anneal-path2) ; exponential cooling
    nil)

(defun anneal-path ()
  (let ((detour-region) (attempts)    ; "attempts" is assigned the
number
      (detour-count 0) (move-prob) ;  of regions in the region list
      (visited))
    (loop
      (setf attempts (segment-index
                  (first (last *winlist*))))
      (setf detour-region (random-region))
(pprint "detour-region:")(pprint detour-region)
      (unless (memberp detour-region visited)
        (detour-at detour-region)
        (cond (*detour-rlist*
               (qdraw)
               (converge *first-win*)
               (draw-final *first-win*)
               (setf *current-path* (store-path *first-win*))
               (setf *accept-path*
              (- *accept-path* 0.1))
               (cond ((or (null *best-path*)
                    (< (first *current-path*)
                     (first *best-path*)))
                     (setf *best-path* *current-path*)
                     (setf *path-list* (append *path-list*
                                  (list *current-path*))))
                   (t
                      (setf move-prob (/ (random 100) 100))
                    (if (or (> (first *current-path*)
                               *infinity*))
                           (> move-prob *accept-path*))
                       (restore-path)))))
      (setf visited (append visited
                            (list detour-region)))) ; end unless
      (incf detour-count)
      (if (>= detour-count attempts)
        (return))) ; end loop
      (draw-best-path *best-path*)
      (print-stored-costs *path-list*))) ; end let, defun

(defun anneal-path2 ()
  (let ((temp (first *best-path*))
      (accepted 0) (prob)
      (detour-region) (visited)
      (delta-c) (decision-point))
    (unless (> *temp-f* temp)
      (loop
      (dotimes (index *l-move*)
        (setf detour-region (random-region))
          (unless (memberp detour-region visited)
        (detour-at detour-region)
        (cond (*detour-rlist*
              (qdraw)
              (converge *first-win*)
              (draw-final *first-win*)
              (setf *current-path* (store-path *first-win*))
              (setf delta-c (- (first *current-path*)
                           (first *best-path*)))
              (cond ((and (< accepted *l-accept*)
                          (< delta-c 0))
                      (incf accepted)
                          (setf *path-list*
                      (append *path-list*
                            (list *best-path*)))
                      (setf *best-path* *current-path*))
                     (t
                      (setf prob
                      (expt *e* (- (/ delta-c temp))))
                      (setf decision-point (/ (random 100) 100))
                      (if (or (> (first *current-path*)
                                *infinity*)
                               (> decision-point prob))
                        (restore-path)))))) ; end cond
        (setf visited (append visited
                              (list detour-region))))) ; end unless,
dotimes
      (setf temp (* temp *reduction-factor*))
      (if (<= temp *temp-f*)
          (return)))) ; end loop, unless
      (draw-best-path *best-path*)
      (print-stored-costs *path-list*))) ;end let, defun

(defun find-path (start goal)
  (let ((start-region (find-region *start*))
      (goal-region  (find-region *goal*))
      (path-cost))
(pprint "start-region:  ")(pprint start-region)
(pprint "goal-region:  ")(pprint goal-region)
    (cond ((eq start-region goal-region)
           (setf *rlist* (list start-region))
         (pprint "*rlist*:  ")(pprint *rlist*)
         (create-window-list start goal)
         (setf path-cost
           (check-direct-path start goal *first-win*))
         (unless (null path-cost)
           (setf (segment-cost *first-win*)
             path-cost)
           (pprint "heading:  ")
           (pprint (rad-to-deg (angle (cons *start*
                                       (list *goal*)))))
           (pprint "path-cost:  ")
           (pprint path-cost)
           (draw-direct *start* *goal* *first-win*))))

    (cond ((null path-cost)
           (setf *rlist* (region-search start goal
                               start-region
                               goal-region))
           (cond (*rlist*
                  (pprint "*rlist*:  ")(pprint *rlist*)
                  (create-window-list start goal)
                (qdraw)
                (converge *first-win*)
                (draw-final *first-win*))
            (t
             (pprint "no solution"))))))) ; end cond, cond let,
defun

(defun check-direct-path (p1 p2 current)
  (let* ((edge-t (cons p1 (list p2)))
         (hdg-t  (angle edge-t)))
    (cond ((inside-rb hdg-t (segment-critical-b *first-win*)
                   (segment-dual-b *first-win*))
         (b-cost p1 p2 nil nil))
        ((inside-pl hdg-t (segment-critical-p *first-win*)
                 (segment-dual-p *first-win*))
         (p-cost (p1 p2 *first-win*)))
        ((inside-pl hdg-t (segment-critical-l *first-win*)
                 (segment-dual-l *first-win*))
         nil)
        ((inside-rb hdg-t (segment-critical-r *first-win*)
                 (segment-dual-r *first-win*))
         nil)
        (t
         (i-cost p1 p2 *first-win*)))))

(defun store-path (init)
  (let ((current init)
      (path-cost 0)
      (temp) (path-list))
    (loop
      (setf temp (make-path
                   :region    (segment-region current)
             :point     (segment-exit-point current)
             :heading   (segment-heading current)
             :cost      (segment-cost current)))
      (setf path-cost (+ path-cost (segment-cost current)))
      (setf path-list (append path-list (list temp)))
      (setf current (segment-next-win current))
      (if (null current)
        (return)))
    (append (list path-cost) path-list)))

(defun print-best-path (path-list)
  (let* ((cost (first path-list))
      (segments (rest path-list)))
    (dolist (element segments)
      (pprint "----------")
      (pprint "point:")   (pprint (path-point element))
      (pprint "heading:") (pprint (path-heading element))
      (pprint "cost:")    (pprint (path-cost element)))
      (pprint "----------")
      (pprint "path-cost:") (pprint cost)
      (pprint "----------")))


(defun print-stored-costs (path-list)
  (pprint "----------")
  (pprint "alternate path costs:")
;  (pprint "----------")
    (cond (path-list
      (dolist (path path-list)
;        (setf segments (rest path))
;        (dolist (element segments)
;       (pprint "----------")
;       (pprint "point:")   (pprint (path-point element))
;       (pprint "heading:") (pprint (path-heading element))
;       (pprint "cost:")    (pprint (path-cost element)))
;        (pprint "----------")
        (pprint "path-cost:")     (pprint (first path))))
       (t
        (pprint "no improvements found"))))
```

```
;**************************************************************
;
;  Filename:  anneal.cl
;
;  By:  HILTON, Cary A.
;
;  Date:  May 91
;
;  Function:  Performs path annealing.
;
;             1.  detours around path segments
;                  with an impermissible heading
;
;             2.  performs random path annealing
;
;             3.  uses A* search from a detour start point
;                 to a detour goal point
;
;             4.  detour path cannot pass through more than
;                 3 regions
;
;**************************************************************

(defvar *detour-rlist* nil)
(defvar *detour-start* nil)
(defvar *detour-goal* nil)
(defvar *upstream-regions* nil)
(defvar *downstream-regions* nil)
(defvar *droot* nil)

(defvar *save-link-begin*)
(defvar *save-link-end*)
(defvar *restore-link-begin*)
(defvar *restore-link-end*)

(defun find-region-in-winlist (target-region)
  (let ((current-window *first-win*))
    (loop
      (if (eq (segment-region current-window)
               target-region)
          (return)
        (setf current-window (segment-next-win
                               current-window))))
     current-window))

(defun find-index-in-winlist (win-num)
  (let ((current-window *first-win*))
    (loop
      (if (eq (segment-index current-window)
               win-num)
          (return)
        (setf current-window (segment-next-win
                               current-window))))
     (segment-region current-window)))
```

```
(defun detour (avoid-region)
  (let ((start-region) (goal-region)
        (detour-start) (detour-goal)
      (avoid-edge)
      (window))
    (setf window (find-region-in-winlist avoid-region))
    (setf detour-start (segment-exit-point
                         (segment-prior-win window)))
    (setf detour-goal  (segment-exit-point
                      (segment-next-win window)))
    (setf start-region (segment-region window))
    (setf goal-region  (segment-region
                      (segment-next-win window)))
    (setf avoid-edge (segment-frontier window))
    (setf *upstream-regions* (get-upstream avoid-region *rlist*))
      (setf  *downstream-regions*  (get-downstream  avoid-region
*rlist*))
    (setf *detour-rlist* (anneal-search detour-start
                                         detour-goal
                                         start-region
                                         goal-region
                                         avoid-edge))))

(defun get-downstream (region rlist)
  (let (downstream)
    (loop
      (if (or (null rlist)
            (eq region (first rlist)))
        (return))
      (setf downstream
        (append downstream (list (first rlist))))
      (setf rlist (rest rlist)))
    downstream))

(defun get-upstream (region rlist)
  (loop
    (if (or (null rlist)
          (eq region (first rlist)))
      (return)
      (setf rlist (rest rlist))))
  (rest rlist))


(defun anneal-search (start goal
                      start-region
                      goal-region
                      avoid-edge)
  (let* ((active-region start-region)
       (prior-edge (cons start (list start)))
         (current-edge) (node-triple)
       (active-node) (active-node-cost)
       (est-to-goal) (cost-tot 0) (temp-node)
       (prior-node start) (prior-region start-region)
       (prior-cost 0)
       (back-path (list start-region)) (rlist))
```

```
;(pprint "detour start point:")
;(pprint start)
;(pprint "detour start region:")
;(pprint start-region)
;(pprint "detour goal point:")
;(pprint goal)
;(pprint "detour goal region:")
;(pprint goal-region)
    (setf *droot* nil)
    (setf current-node (make-node
                        :point         start
                               :edge          nil
                        :region        start-region
                        :chain         nil
                               :cost          0
                          :cost-est      0
                             :prior         nil
                          :next          nil))

    (loop
      (dolist (element (polygon-alist (eval active-region)))
       (unless (or  (> (length (node-chain current-node)) 2)
               (and (eq element prior-region)
                  (not (eq start-region goal-region)))
               (virtual-obstacle element)
                      (and (memberp element (node-chain current-
node))
                  (not (eq start-region goal-region)))
               (memberp element *downstream-regions*))

        (setf current-edge (find-edge (eval active-region)
                               (eval element)))
        (setf current-edge
          (cons (eval (first current-edge))
             (list (eval (second current-edge)))))
        (unless (or (null (first current-edge))
                 (null (valid-entry prior-edge current-edge
                                      active-region))
                 (equal current-edge avoid-edge))

            (setf node-triple (append
                          (list (mid-point (first current-edge)
                                (second current-edge)))
                          (list (first current-edge))
                          (list (second current-edge))))

            (dolist (active-node node-triple)
          (setf active-node-cost
            (+ (w-distance prior-node active-node
                     (polygon-weight (eval active-region)))
             prior-cost))
            (draw-dotted (cons active-node (list prior-node)))
          (setf est-to-goal (distance active-node goal))
            (setf cost-tot    est-to-goal)
;         (setf cost-tot    (+ active-node-cost est-to-goal))
```

```
        (setf temp-node (make-node
                     :point        active-node
                            :edge         current-edge
                     :region       element
                     :chain        back-path
                            :cost         active-node-cost
                       :cost-est      cost-tot
                          :prior        nil
                       :next         nil))
         (setf *droot* (insert-node temp-node *droot*))) ; end
dolist


)))  ;end unless, unless, unless, dolist


       (if (null *droot*)
         (return)
         (setf current-node *droot*))
       (setf back-path  (append (node-chain current-node)
                               (list (node-region current-node))))
     (setf prior-cost (node-cost current-node))
      (if (memberp (node-region current-node) *upstream-regions*)
        (return))
     (setf prior-edge (node-edge current-node))
     (setf *droot* (node-next *droot*))
     (setf prior-region active-region)
     (setf prior-node (node-point current-node))
     (setf active-region (node-region current-node)))  ; end loop

     (if (eq (node-region current-node) goal-region)
       (draw-dotted (cons (node-point current-node)
                         (list goal))))
     (setf rlist (append (node-chain current-node)
                      (list (node-region current-node))))
     (if (memberp (first (last rlist))
                  *upstream-regions*)
         rlist
       nil)))

(defun detour-at (detour-segment)
  (let ((detour-rlist) (detour-winlist)
      (detour-start) (detour-goal)
      (index) (save-frontier) (save-length)
      (detour-region detour-segment))
    (cond (detour-segment
        (setf detour-rlist (detour detour-region))
          (cond (detour-rlist
              (setf detour-start (find-region-in-winlist detour-
region))
          (setf detour-goal (find-region-in-winlist
                          (first (last detour-rlist))))
            (setf save-frontier (segment-frontier detour-goal))
             (setf save-length   (segment-frontier-length detour-
goal))
```

```
          (setf *detour-winlist*
          (change-struct
           (segment-exit-point detour-start)
           (segment-exit-point detour-goal)
           detour-rlist))
         (double-link  (first  *detour-winlist*)  (rest  *detour-
winlist*))
           (setf (segment-next-win (first (last *detour-winlist*)))
          nil)
           (setf (segment-frontier (first (last *detour-winlist*)))
          save-frontier)
         (setf  (segment-frontier-length  (first  (last  *detour-
winlist*)))
          save-length)
           (setf *save-link-begin* detour-start)
            (setf *restore-link-begin* (segment-prior-win detour-
start))
        (setf (segment-next-win *restore-link-begin*)
          (second *detour-winlist*))
        (setf (segment-prior-win (second *detour-winlist*))
          *restore-link-begin*)
           (unless (null (segment-next-win detour-goal))
          (setf *save-link-end* detour-goal)
          (setf *restore-link-end*
            (segment-next-win detour-goal))
          (setf (segment-prior-win
              *restore-link-end*)
            (first (last *detour-winlist*)))
        (setf (segment-next-win (first (last *detour-winlist*)))
          *restore-link-end*))
           (renumber-indices)))))))

(defun renumber-indices ()
  (let ((current *init-win*) (index 0))
    (setf *first-win*
      (segment-next-win  *init-win*))
    (loop
      (setf (segment-index current) index)
      (setf current (segment-next-win current))
      (incf index)
      (if (null current)
        (return))))) ;end loop, let, defun

(defun random-region ()
  (let ((detour-num) (detour-region)
      (win-num))
    (setf *win-num*
      (segment-index (first (last *winlist*))))
    (setf detour-num (random *win-num*))
    (if (< detour-num 1)
      (setf detour-num 1))
    (find-index-in-winlist detour-num)))

(defun restore-path ()
  (setf (segment-next-win *restore-link-begin*)
    *save-link-begin*)
  (setf (segment-prior-win *save-link-begin*)
    *restore-link-begin*)
  (if *restore-link-end*
    (setf (segment-prior-win *restore-link-end*)
      *save-link-end*))
  (setf (segment-next-win *save-link-end*)
    *restore-link-end*)
  (setf *first-win* (segment-next-win
                  *init-win*))
  (renumber-indices)
 nil)
```

```
;****************************************************************
;
;  Filename:  converge.cl
;
;  By:  HILTON, Cary A.
;
;  Date:  May 91
;
;   Function:   Control structure for the golden ratio search
routine.
;
;              1)  setf tolerances which determine when to
;                  stop the search
;
;              2)  computes current path cost
;
;***************************************************************

(defvar *edge-tolerance* 12)     ;  edge length during golden
                                 ;   ratio search
(defvar *point-tolerance* 6)     ;  crossing point displacement
                                 ;   used in "converge"
(defvar *precision*              ;  halt search if improvement of
                                 ;   current cost is less than this
value
(setf *lcount* 6)                   ;  Max number of outer loops
allowed
                                 ;   during "converge"

;***************************************************************

(defun converge (current)
  (let ((anchored 1) (count 0)
      (current-cost 0) (prev-cost *infinity*))
    (loop
      (setf anchored 1)
    (loop
      (unless (or (null (segment-next-win current))
                  (= (segment-frontier-length current) 0))
;         (if (and (= (first (segment-exit-point current))
;                        (first (segment-exit-point (segment-next-win
current))))
;                (= (second (segment-exit-point current))
;                     (second (segment-exit-point (segment-next-win
current)))))
;             (snip (segment-next-win current)))
         (adjust-crossing current)
       (if (> (segment-displ current) *point-tolerance*)
           (setf anchored 0))) ; end unless

      (if (null (segment-next-win current))
        (return)
      (setf current (segment-next-win current)))) ;end loop
      (incf count)
      (setf prev-cost current-cost)
      (setf current-cost (calc-path-cost *first-win*))
(pprint "path cost")(pprint current-cost)
      (if (or (plusp anchored)
            (trivial-improvement
                prev-cost current-cost)
            (< *lcount* count))
        (return)
      (setf current *first-win*)))    ; end loop
    (pprint "*lcount*")
    (pprint count)))

(defun trivial-improvement (c1 c2)
  (let ((diff (- c1 c2)))
    (if (or (> c2 c1)
            (< (/ diff c1)
               *precision*))
        t
      nil)))

(defun calc-path-cost (element)
  (let ((tot 0))
    (loop
      (setf tot (+ tot (segment-cost element)))
      (setf element (segment-next-win element))
      (if (null element)
        (return)))
    tot))

;  "snip" removes a window from
;   the window list.  It is
;   not used in current version.

(defun snip (element)
(pprint "snipping:  ")
(pprint (segment-region element))
   (if (eq element *first-win*)
       (setf *first-win* (segment-next-win *first-win*)))
  (if (not (null (segment-prior-win element)))
    (setf (segment-next-win (segment-prior-win element))
        (segment-next-win element)))
  (if (not (null (segment-next-win element)))
    (setf (segment-prior-win (segment-next-win element))
          (segment-prior-win element))))
```

```
;*****************************************************************
;
;  Filename:  costf.cl
;
;  By:  HILTON, Cary A.
;
;  Date:  May 91
;
;  Function:  Contains cost functions for all possible
;              combinations of heading ranges between
;               two regions.
;
;             1.  gets pre-point, pre-range, frontier,
;                 post-point, post-range, current,
;                 then branches to appropriate cost function.
;
;              2.  returns cost, crossing point, and heading range
combination
;
;****************************************************************

(defvar *interval* 3)  ;  used to space points along a segment.


(defun costf (from pre-range edge to post-range current loop-flag)
  (cond ((or (eq pre-range  's)
             (eq post-range 's))
            (hi-search from edge to current loop-flag pre-range post-
range))
        ((eq pre-range 'i)
         (cond ((eq post-range 'i)
              (ii-search from edge to current loop-flag))
               ((eq post-range 'p)
              (ip-search from edge to current loop-flag))
               ((eq post-range 'b)
              (ib-search from edge to current loop-flag))))
        ((eq pre-range 'p)
         (cond ((eq post-range 'i)
              (pi-search from edge to current loop-flag))
               ((eq post-range 'p)
              (pp-search from edge to current loop-flag))
               ((eq post-range 'b)
              (pb-search from edge to current loop-flag))))
        ((eq pre-range 'b)
         (cond ((eq post-range 'i)
              (bi-search from edge to current loop-flag))
               ((eq post-range 'p)
              (bp-search from edge to current loop-flag))
               ((eq post-range 'b)
              (bb-search from edge to current loop-flag))))))

;*********************************************

; high-cost search
```

```
(defun hi-search (from edge to current loop-flag pre-range post-
range)
  (let* ((x1) (y1) (x2) (y2)
        (p1) (p2) (p3) (p4)
          (dx) (dy) (ranges)
        (pre1)  (pre2)  (pre3)  (pre4)
        (post1) (post2) (post3) (post4)
        (edge-length (+ *edge-tolerance* 1))
        (c1) (c2) (c3) (c4) (c-min))

          (loop
          (setf p1 (first edge))
          (setf p4 (second edge))
            (setf x1 (first  p1))
          (setf y1 (second p1))
          (setf x2 (first  p4))
          (setf y2 (second p4))
            (setf dx (fround (/ (- x2 x1) *interval*)))
          (setf dy (fround (/ (- y2 y1) *interval*)))
          (setf p2 (cons (+ x1 dx)
                (list (+ y1 dy))))
          (setf p3 (cons (- x2 dx)
                (list (- y2 dy))))

            (cond ((eq pre-range 's)
                (setf pre1  (hi-cost from p1 current))
                (setf pre2  (hi-cost from p2 current))
                  (setf pre3  (hi-cost from p3 current))
                (setf pre4  (hi-cost from p4 current)))
             (t
                (setf pre1  (i-cost from p1 current))
                (setf pre2  (i-cost from p2 current))
                  (setf pre3  (i-cost from p3 current))
                (setf pre4  (i-cost from p4 current))))

            (cond ((eq post-range 's)
                  (setf post1 (hi-cost p1   to (segment-next-win
current)))
                  (setf post2 (hi-cost p2   to (segment-next-win
current)))
                  (setf post3 (hi-cost p3   to (segment-next-win
current)))
                  (setf post4 (hi-cost p4   to (segment-next-win
current))))
                 (t
                  (setf post1 (i-cost p1    to (segment-next-win
current)))
                  (setf post2 (i-cost p2    to (segment-next-win
current)))
                  (setf post3 (i-cost p3    to (segment-next-win
current)))
                  (setf post4 (i-cost p4    to (segment-next-win
current)))))

          (setf c1 (+ pre1 post1))
        (setf c2 (+ pre2 post2))
          (setf c3 (+ pre3 post3))
        (setf c4 (+ pre4 post4))
          (setf c-min (min c1 c2 c3 c4))
        (if (< c2 c3)
            (setf edge (cons p1 (list p3)))
          (setf edge (cons p2 (list p4))))
        (setf edge-length (distance (first edge)
                                    (second edge)))
        (if (or (null loop-flag)
                (< edge-length *edge-tolerance*))
            (return)))

    (setf ranges (cons pre-range (list post-range)))
    (cond ((= c-min c1)
        (append (list c1) (list pre1) (list post1) (list p1) (list
ranges)))
        ((= c-min c2)
        (append (list c2) (list pre2) (list post2) (list p2) (list
ranges)))
        ((= c-min c3)
        (append (list c3) (list pre3) (list post3) (list p3) (list
ranges)))
        (t
            (append (list c4) (list pre4) (list post4) (list p4)
(list ranges))))))



;************************************
; isotropic-to-isotropic search

(defun ii-search (from edge to current loop-flag)
  (let* ((crossing-pt)
       (path-leg)
       (pre1)  (pre2)
       (post1) (post2)
       (c1)  (c2))
            (setf path-leg (cons from (list to)))
            (setf crossing-pt (find-int path-leg edge))
                (cond ((null crossing-pt)
                  (setf pre1 (i-cost from
                                 (first edge)
                                 current))
                 (setf post1 (i-cost (first edge)
                                     to
                                     (segment-next-win
                                      current)))
                  (setf pre2 (i-cost from
                                 (second edge)
                                 current))
                  (setf post2  (i-cost (second edge)
                                  to
                                  (segment-next-win
                                current)))
               (setf c1 (+ pre1 post1))
               (setf c2 (+ pre2 post2))

               (if (< c1 c2)
          (append (list c1) (list pre1)
             (list post1) (list (first edge)) (list '(i i)))

          (append (list c2) (list pre2)
             (list post2) (list (second edge)) (list '(i i)))))
              (t
               (setf pre1  (i-cost from
                            crossing-pt
                            current))
               (setf post1 (i-cost crossing-pt
                            to
                             (segment-next-win
                              current)))
               (setf c1 (+ pre1 post1))
          (append (list c1) (list pre1)
             (list post1) (list crossing-pt) (list '(i i)))))))

;***********************************
; isotropic-to-power search

(defun ip-search (from edge to current loop-flag)
  (let* ((x1) (y1) (x2) (y2)
       (p1) (p2) (p3) (p4)
         (dx) (dy)
       (pre1)  (pre2)  (pre3)  (pre4)
       (post1) (post2) (post3) (post4)
       (edge-length (+ *edge-tolerance* 1))
       (new-pt (segment-exit-point current))
       (c1) (c2) (c3) (c4) (c-min))

         (loop
        (setf p1 (first edge))
        (setf p4 (second edge))
          (setf x1 (first  p1))
        (setf y1 (second p1))
        (setf x2 (first  p4))
        (setf y2 (second p4))
          (setf dx (fround (/ (- x2 x1) *interval*)))
        (setf dy (fround (/ (- y2 y1) *interval*)))
        (setf p2 (cons (+ x1 dx)
              (list (+ y1 dy))))
        (setf p3 (cons (- x2 dx)
              (list (- y2 dy))))
        (setf pre1  (i-cost from p1 current))
        (setf post1 (p-cost p1   to (segment-next-win current)))
        (setf pre2  (i-cost from p2 current))
        (setf post2 (p-cost p2   to (segment-next-win current)))
          (setf pre3  (i-cost from p3 current))
        (setf post3 (p-cost p3   to (segment-next-win current)))
          (setf pre4  (i-cost from p4 current))
        (setf post4 (p-cost p4   to (segment-next-win current)))
        (setf c1 (+ pre1 post1))
        (setf c2 (+ pre2 post2))
          (setf c3 (+ pre3 post3))
        (setf c4 (+ pre4 post4))
          (setf c-min (min c1 c2 c3 c4))
        (if (< c2 c3)
            (setf edge (cons p1 (list p3)))
          (setf edge (cons p2 (list p4))))
        (setf edge-length (distance (first edge)
                                    (second edge)))
        (if (or (null loop-flag)
              (< edge-length *edge-tolerance*))
            (return)))

    (cond ((= c-min c1)
        (append (list c1) (list pre1) (list post1) (list p1) (list
'(i p))))
       ((= c-min c2)
        (append (list c2) (list pre2) (list post2) (list p2) (list
'(i p))))
       ((= c-min c3)
        (append (list c3) (list pre3) (list post3) (list p3) (list
'(i p))))
       (t
            (append (list c4) (list pre4) (list post4) (list p4)
(list '(i p)))))))



;************************************
; isotropic-to-braking search

(defun ib-search (from edge to current loop-flag)
  (let* ((x1) (y1) (x2) (y2)
       (p1) (p2) (p3) (p4)
         (dx) (dy)
       (pre1)  (pre2)  (pre3)  (pre4)
       (post1) (post2) (post3) (post4)
       (edge-length (+ *edge-tolerance* 1))
       (c1) (c2) (c3) (c4) (c-min))
         (loop
        (setf p1 (first edge))
        (setf p4 (second edge))
          (setf x1 (first  p1))
        (setf y1 (second p1))
        (setf x2 (first  p4))
        (setf y2 (second p4))
          (setf dx (fround (/ (- x2 x1) *interval*)))
        (setf dy (fround (/ (- y2 y1) *interval*)))
        (setf p2 (cons (+ x1 dx)
               (list (+ y1 dy))))
        (setf p3 (cons (- x2 dx)
               (list (- y2 dy))))
        (setf pre1  (i-cost from p1 current))
        (setf post1 (b-cost p1   to
                                  (segment-frontier current)
                           (segment-frontier
                             (segment-next-win current))))
        (setf pre2  (i-cost from p2 current))
        (setf post2 (b-cost p2   to
                                  (segment-frontier current)
                           (segment-frontier
                             (segment-next-win current))))
          (setf pre3  (i-cost from p3 current))
        (setf post3 (b-cost p3   to
                                  (segment-frontier current)
                           (segment-frontier
                             (segment-next-win current))))
          (setf pre4  (i-cost from p4 current))
        (setf post4 (b-cost p4   to
                                  (segment-frontier current)
                           (segment-frontier
                             (segment-next-win current))))
        (setf c1 (+ pre1 post1))
        (setf c2 (+ pre2 post2))
          (setf c3 (+ pre3 post3))
        (setf c4 (+ pre4 post4))
          (setf c-min (min c1 c2 c3 c4))
        (if (< c2 c3)
            (setf edge (cons p1 (list p3)))
          (setf edge (cons p2 (list p4))))
        (setf edge-length (distance (first edge)
                                    (second edge)))
        (if (or (null loop-flag)
              (< edge-length *edge-tolerance*))
            (return)))

    (cond ((= c-min c1)
        (append (list c1) (list pre1) (list post1) (list p1) (list
'(i b))))
       ((= c-min c2)
        (append (list c2) (list pre2) (list post2) (list p2) (list
'(i b))))
       ((= c-min c3)
        (append (list c3) (list pre3) (list post3) (list p3) (list
'(i b))))
       (t
           (append (list c4) (list pre4) (list post4) (list p4)
(list '(i b)))))))


;************************************
; power-to-isotropic search

(defun pi-search (from edge to current loop-flag)
  (let* ((x1) (y1) (x2) (y2)
       (p1) (p2) (p3) (p4)
         (dx) (dy)
       (pre1)  (pre2)  (pre3)  (pre4)
       (post1) (post2) (post3) (post4)
       (edge-length (+ *edge-tolerance* 1))
       (c1) (c2) (c3) (c4) (c-min))

         (loop
         (setf p1 (first edge))
         (setf p4 (second edge))
           (setf x1 (first  p1))
         (setf y1 (second p1))
         (setf x2 (first  p4))
         (setf y2 (second p4))
           (setf dx (fround (/ (- x2 x1) *interval*)))
         (setf dy (fround (/ (- y2 y1) *interval*)))
         (setf p2 (cons (+ x1 dx)
               (list (+ y1 dy))))
         (setf p3 (cons (- x2 dx)
               (list (- y2 dy))))       (setf pre1  (p-cost from p1
current))
       (setf post1 (i-cost p1   to (segment-next-win current)))
       (setf pre2  (p-cost from p2 current))
       (setf post2 (i-cost p2   to (segment-next-win current)))
         (setf pre3  (p-cost from p3 current))
       (setf post3 (i-cost p3   to (segment-next-win current)))
         (setf pre4  (p-cost from p4 current))
       (setf post4 (i-cost p4   to (segment-next-win current)))
       (setf c1 (+ pre1 post1))
       (setf c2 (+ pre2 post2))
         (setf c3 (+ pre3 post3))
       (setf c4 (+ pre4 post4))
           (setf c-min (min c1 c2 c3 c4))
         (if (< c2 c3)
             (setf edge (cons p1 (list p3)))
           (setf edge (cons p2 (list p4))))
         (setf edge-length (distance (first edge)
                                     (second edge)))
         (if (or (null loop-flag)
               (< edge-length *edge-tolerance*))
             (return)))

    (cond ((= c-min c1)
         (append (list c1) (list pre1) (list post1) (list p1) (list
'(p i))))
        ((= c-min c2)
         (append (list c2) (list pre2) (list post2) (list p2) (list
'(p i))))
        ((= c-min c3)
         (append (list c3) (list pre3) (list post3) (list p3) (list
'(p i))))
        (t
             (append (list c4) (list pre4) (list post4) (list p4)
(list '(p i)))))))

;***********************************
; power-to-power search
```

```
(defun pp-search (from edge to current loop-flag)
  (let* ((x1) (y1) (x2) (y2)
       (p1) (p2) (p3) (p4)
         (dx) (dy)
       (pre1)  (pre2)  (pre3)  (pre4)
       (post1) (post2) (post3) (post4)
       (edge-length (+ *edge-tolerance* 1))
       (c1) (c2) (c3) (c4) (c-min))

         (loop
         (setf p1 (first edge))
         (setf p4 (second edge))
           (setf x1 (first  p1))
         (setf y1 (second p1))
         (setf x2 (first  p4))
         (setf y2 (second p4))
           (setf dx (fround (/ (- x2 x1) *interval*)))
         (setf dy (fround (/ (- y2 y1) *interval*)))
         (setf p2 (cons (+ x1 dx)
                (list (+ y1 dy))))
         (setf p3 (cons (- x2 dx)
                (list (- y2 dy))))        (setf pre1  (p-cost from
p1 current))
         (setf post1 (p-cost p1   to (segment-next-win current)))
         (setf pre2  (p-cost from p2 current))
         (setf post2 (p-cost p2   to (segment-next-win current)))
           (setf pre3  (p-cost from p3 current))
         (setf post3 (p-cost p3   to (segment-next-win current)))
           (setf pre4  (p-cost from p4 current))
         (setf post4 (p-cost p4   to (segment-next-win current)))
         (setf c1 (+ pre1 post1))
         (setf c2 (+ pre2 post2))
           (setf c3 (+ pre3 post3))
         (setf c4 (+ pre4 post4))
           (setf c-min (min c1 c2 c3 c4))
         (if (< c2 c3)
             (setf edge (cons p1 (list p3)))
           (setf edge (cons p2 (list p4))))
         (setf edge-length (distance (first edge)
                                (second edge)))
         (if (or (null loop-flag)
               (< edge-length *edge-tolerance*))
             (return)))

    (cond ((= c-min c1)
         (append (list c1) (list pre1) (list post1) (list p1) (list
'(p p))))
        ((= c-min c2)
         (append (list c2) (list pre2) (list post2) (list p2) (list
'(p p))))
        ((= c-min c3)
         (append (list c3) (list pre3) (list post3) (list p3) (list
'(p p))))
        (t
            (append (list c4) (list pre4) (list post4) (list p4)
(list '(p p)))))))


;**********************************
; power-to-braking search

(defun pb-search (from edge to current loop-flag)
  (let* ((x1) (y1) (x2) (y2)
        (p1) (p2) (p3) (p4)
          (dx) (dy)
        (pre1)  (pre2)  (pre3)  (pre4)
        (post1) (post2) (post3) (post4)
        (edge-length (+ *edge-tolerance* 1))
        (c1) (c2) (c3) (c4) (c-min))

          (loop
          (setf p1 (first edge))
          (setf p4 (second edge))
            (setf x1 (first  p1))
          (setf y1 (second p1))
          (setf x2 (first  p4))
          (setf y2 (second p4))
            (setf dx (fround (/ (- x2 x1) *interval*)))
          (setf dy (fround (/ (- y2 y1) *interval*)))
          (setf p2 (cons (+ x1 dx)
                (list (+ y1 dy))))
          (setf p3 (cons (- x2 dx)
                (list (- y2 dy))))
          (setf pre1  (p-cost from p1 current))
        (setf post1 (b-cost p1   to
                                   (segment-frontier current)
                             (segment-frontier
                               (segment-next-win current))))
        (setf pre2  (p-cost from p2 current))
        (setf post2 (b-cost p2   to
                             (segment-frontier current)
                             (segment-frontier
                               (segment-next-win current))))
          (setf pre3  (p-cost from p3 current))
        (setf post3 (b-cost p3   to
                             (segment-frontier current)
                             (segment-frontier
                               (segment-next-win current))))
          (setf pre4  (p-cost from p4 current))
        (setf post4 (b-cost p4   to
                             (segment-frontier current)
                             (segment-frontier
                               (segment-next-win current))))
        (setf c1 (+ pre1 post1))
        (setf c2 (+ pre2 post2))
          (setf c3 (+ pre3 post3))
          (setf c4 (+ pre4 post4))
            (setf c-min (min c1 c2 c3 c4))
          (if (< c2 c3)
            (setf edge (cons p1 (list p3)))
          (setf edge (cons p2 (list p4))))
        (setf edge-length (distance (first edge)
                                    (second edge)))
        (if (or (null loop-flag)
              (< edge-length *edge-tolerance*))
            (return)))

    (cond ((= c-min c1)
        (append (list c1) (list pre1) (list post1) (list p1) (list
'(p b))))
       ((= c-min c2)
        (append (list c2) (list pre2) (list post2) (list p2) (list
'(p b))))
       ((= c-min c3)
        (append (list c3) (list pre3) (list post3) (list p3) (list
'(p b))))
       (t
            (append (list c4) (list pre4) (list post4) (list p4)
(list '(p b)))))))

;************************************
; braking-to-isotropic search

(defun bi-search (from edge to current loop-flag)
  (let* ((x1) (y1) (x2) (y2)
       (p1) (p2) (p3) (p4)
         (dx) (dy)
       (pre1)  (pre2)  (pre3)  (pre4)
       (post1) (post2) (post3) (post4)
       (edge-length (+ *edge-tolerance* 1))
       (c1) (c2) (c3) (c4) (c-min))
         (loop
        (setf p1 (first edge))
        (setf p4 (second edge))
          (setf x1 (first  p1))
        (setf y1 (second p1))
        (setf x2 (first  p4))
        (setf y2 (second p4))
          (setf dx (fround (/ (- x2 x1) *interval*)))
        (setf dy (fround (/ (- y2 y1) *interval*)))
        (setf p2 (cons (+ x1 dx)
              (list (+ y1 dy))))
        (setf p3 (cons (- x2 dx)
              (list (- y2 dy))))
        (setf pre1  (b-cost from p1
                    (segment-frontier
                      (segment-prior-win current))
                    (segment-frontier current)))
       (setf post1 (i-cost p1   to (segment-next-win current)))
       (setf pre2  (b-cost from p2
                    (segment-frontier
                      (segment-prior-win current))
                        (segment-frontier current)))
       (setf post2 (i-cost p2   to (segment-next-win current)))
```

```
          (setf pre3  (b-cost from p3
                                  (segment-frontier (segment-prior-win
current))
                            (segment-frontier current)))
        (setf post3 (i-cost p3   to (segment-next-win current)))
          (setf pre4  (b-cost from p4
                                  (segment-frontier (segment-prior-win
current))
                            (segment-frontier current)))
        (setf post4 (i-cost p4   to (segment-next-win current)))
        (setf c1 (+ pre1 post1))
        (setf c2 (+ pre2 post2))
          (setf c3 (+ pre3 post3))
          (setf c4 (+ pre4 post4))
            (setf c-min (min c1 c2 c3 c4))
          (if (< c2 c3)
              (setf edge (cons p1 (list p3)))
            (setf edge (cons p2 (list p4))))
          (setf edge-length (distance (first edge)
                                      (second edge)))
          (if (or (null loop-flag)
                (< edge-length *edge-tolerance*))
              (return)))
     (cond ((= c-min c1)
          (append (list c1) (list pre1) (list post1) (list p1) (list
'(b i))))
         ((= c-min c2)
          (append (list c2) (list pre2) (list post2) (list p2) (list
'(b i))))
         ((= c-min c3)
          (append (list c3) (list pre3) (list post3) (list p3) (list
'(b i))))
         (t
              (append (list c4) (list pre4) (list post4) (list p4)
(list '(b i)))))))


;***********************************
; braking-to-power search

(defun bp-search (from edge to current loop-flag)
  (let* ((x1) (y1) (x2) (y2)
        (p1) (p2) (p3) (p4)
          (dx) (dy)
        (pre1)  (pre2)  (pre3)  (pre4)
        (post1) (post2) (post3) (post4)
        (edge-length (+ *edge-tolerance* 1))
        (new-pt (segment-exit-point current))
        (c1) (c2) (c3) (c4) (c-min))

          (loop
          (setf p1 (first edge))
          (setf p4 (second edge))
            (setf x1 (first  p1))
          (setf y1 (second p1))
        (setf x2 (first  p4))
        (setf y2 (second p4))
          (setf dx (fround (/ (- x2 x1) *interval*)))
        (setf dy (fround (/ (- y2 y1) *interval*)))
        (setf p2 (cons (+ x1 dx)
              (list (+ y1 dy))))
        (setf p3 (cons (- x2 dx)
              (list (- y2 dy))))
        (setf pre1  (b-cost from p1
                     (segment-frontier
                       (segment-prior-win current))
                     (segment-frontier current)))
      (setf post1 (p-cost p1   to (segment-next-win current)))
      (setf pre2  (b-cost from p2
                     (segment-frontier
                       (segment-prior-win current))
                     (segment-frontier current)))
      (setf post2 (p-cost p2   to (segment-next-win current)))
        (setf pre3  (b-cost from p3
                     (segment-frontier
                       (segment-prior-win current))
                     (segment-frontier current)))
      (setf post3 (p-cost p3   to (segment-next-win current)))
        (setf pre4  (b-cost from p4
                     (segment-frontier
                       (segment-prior-win current))
                     (segment-frontier current)))
      (setf post4 (p-cost p4   to (segment-next-win current)))
      (setf c1 (+ pre1 post1))
      (setf c2 (+ pre2 post2))
        (setf c3 (+ pre3 post3))
        (setf c4 (+ pre4 post4))
          (setf c-min (min c1 c2 c3 c4))
        (if (< c2 c3)
            (setf edge (cons p1 (list p3)))
          (setf edge (cons p2 (list p4))))
        (setf edge-length (distance (first edge)
                                    (second edge)))
        (if (or (null loop-flag)
              (< edge-length *edge-tolerance*))
            (return)))

    (cond ((= c-min c1)
        (append (list c1) (list pre1) (list post1) (list p1) (list
'(b p))))
       ((= c-min c2)
        (append (list c2) (list pre2) (list post2) (list p2) (list
'(b p))))
       ((= c-min c3)
        (append (list c3) (list pre3) (list post3) (list p3) (list
'(b p))))
       (t
            (append (list c4) (list pre4) (list post4) (list p4)
(list '(b p)))))))
```

```
;**********************************
; braking-to-braking search

(defun bb-search (from edge to current loop-flag)
  (let* ((x1) (y1) (x2) (y2)
       (p1 (first  (segment-frontier current)))
       (p4 (second (segment-frontier current)))
       (p2) (p3)
         (dx) (dy)
         (path-leg)
       (dist1) (dist2)
       (pre1)  (pre2)  (pre3)  (pre4)
       (post1) (post2) (post3) (post4)
       (edge-length (+ *edge-tolerance* 1))
       (c1) (c2) (c3) (c4) (c-min))
      (cond ((equal (third p1)
              (third p4))
          (setf path-leg (cons from (list to)))
          (setf p1 (find-int path-leg edge))
            (cond ((null p1)
              (setf dist1 (+ (i-cost from
                              (first edge)
                              current)
                         (i-cost (first edge)
                              to
                              (segment-next-win
                              current))))
              (setf dist2 (+ (i-cost from
                              (second edge)
                              current)
                         (i-cost (second edge)
                              to
                              (segment-next-win
                              current))))
                 (if (< dist1 dist2)
                   (setf p1 (first edge))
                 (setf p1 (second edge)))))

            (setf pre1  (b-cost from p1
                            (segment-frontier
                             (segment-prior-win current))
                            (segment-frontier current)))
            (setf post1 (b-cost p1   to
                            (segment-frontier current)
                              (segment-frontier
                             (segment-next-win current))))
          (setf c1 (+ pre1 post1))
            (append (list c1) (list pre1)
               (list post1) (list p1) (list '(b b))))
           (t
        (loop
        (setf p1 (first edge))
        (setf p4 (second edge))
          (setf x1 (first  p1))
        (setf y1 (second p1))
        (setf x2 (first  p4))
        (setf y2 (second p4))
          (setf dx (fround (/ (- x2 x1) *interval*)))
        (setf dy (fround (/ (- y2 y1) *interval*)))
        (setf p2 (cons (+ x1 dx)
               (list (+ y1 dy))))
        (setf p3 (cons (- x2 dx)
               (list (- y2 dy))))
        (setf pre1  (b-cost from p1
                           (segment-frontier  (segment-prior-win
current))
                       (segment-frontier current)))
        (setf post1 (b-cost p1   to
                       (segment-frontier current)
                            (segment-frontier  (segment-next-win
current))))
        (setf pre2  (b-cost from p2
                           (segment-frontier  (segment-prior-win
current))
                       (segment-frontier current)))
        (setf post2 (b-cost p2   to
                       (segment-frontier current)
                            (segment-frontier  (segment-next-win
current))))
          (setf pre3  (b-cost from p3
                           (segment-frontier  (segment-prior-win
current))
                       (segment-frontier current)))
        (setf post3 (b-cost p3   to
                       (segment-frontier current)
                            (segment-frontier  (segment-next-win
current))))
          (setf pre4  (b-cost from p4
                           (segment-frontier  (segment-prior-win
current))
                       (segment-frontier current)))
        (setf post4 (b-cost p4   to
                       (segment-frontier current)
                            (segment-frontier  (segment-next-win
current))))
        (setf c1 (+ pre1 post1))
        (setf c2 (+ pre2 post2))
        (setf c3 (+ pre3 post3))
        (setf c4 (+ pre4 post4))
          (setf c-min (min c1 c2 c3 c4))
        (if (< c2 c3)
            (setf edge (cons p1 (list p3)))
          (setf edge (cons p2 (list p4))))
        (setf edge-length (distance (first edge)
                                    (second edge)))
        (if (or (null loop-flag)
              (< edge-length *edge-tolerance*))
            (return)))
```

```
    (cond ((= c-min c1)
           (append (list c1) (list pre1) (list post1) (list p1) (list
'(b b))))
          ((= c-min c2)
           (append (list c2) (list pre2) (list post2) (list p2) (list
'(b b))))
          ((= c-min c3)
           (append (list c3) (list pre3) (list post3) (list p3) (list
'(b b))))
          (t
                (append (list c4) (list pre4) (list post4) (list p4)
(list '(b b)))))))))

;***********************************

;  If no intersection, assume
;   heading is on a critical
;   angle and return mid-point
;   of segment as switch-point.

(defun switch-point (p1 p2 hdg1 hdg2)
  (let ((ray1 (calc-ray p1 hdg1))
        (ray2 (calc-ray p2 hdg2))
       (int-pt))
      (setf int-pt (line-int ray1 ray2))
      (if (null int-pt)
        (mid-point p1 p2)
        int-pt)))

;***********************************

(defun i-cost (p1 p2 current)
       (w-distance p1 p2 (segment-weight current)))

(defun p-cost (p1 p2 current)
  (let ((sw-point  (switch-point p1 p2 (segment-critical-p current)
                                 (+ pi (segment-dual-p current)))))
         (+ (i-cost p1 sw-point current)
          (i-cost sw-point p2 current))))
;  the function switch-point is in "geometry.cl"


; Note b-cost gets different arguments than i-cost, p-cost

(defun b-cost (p1 p2 edge1 edge2)
  (let ((elev1) (elev2))
    (cond ((third p1)
           (setf elev1 (third p1)))
          ((equal p1 *start*)
           (setf elev1 (find-elev p1 (first *rlist*))))
          (t
           (setf elev1 (calc-elev p1 edge1))))
    (cond ((third p2)
           (setf elev2 (third p2)))
           ((equal p2 *goal*)
          (setf elev2 (find-elev p2 (first (last *rlist*)))))
         (t
          (setf elev2 (calc-elev p2 edge2))))
    (abs (- elev1 elev2))))


(defun hi-cost (p1 p2 current)
  (if (and (equal (first p1)  (first p2))
           (equal (second p1) (second p2)))
      0
    (+ (w-distance p1 p2 (segment-weight current))
         *infinity*)))
```

```
;*****************************************************************
;
;  Filename:  drawf.cl
;
;  By:  HILTON, Cary A.
;
;  Date:  May 91
;
;  Function:  Various drawing functions
;
;             1.  draws map, path segments, heading ranges
;
;             2.  computes segment headings, segment costs
;
;****************************************************************


(defvar *bad-segments* nil)  ; list of path segments
                             ;  with impermissible headings

(defun draw-regions (rlist)
  (let ((polygon) (vlist))
  (dolist (element rlist)
    (setf polygon (eval element))
    (setf vlist (polygon-vlist polygon))
    (draw-edge (first vlist)
           (first (last vlist)))
    (loop
      (if (null (rest vlist))
        (return))
      (draw-edge (first  vlist)
            (second vlist))
      (setf vlist (rest vlist))))))

(defun draw-region (region)
  (let* ((r (eval region))
       (vlist  (polygon-vlist r)))
    (draw-edge (first vlist)
           (first (last vlist)))
    (loop
      (if (null (rest vlist))
        (return))
      (draw-edge (first  vlist)
            (second vlist))
      (setf vlist (rest vlist)))))


(defun draw-edge (xyz1 xyz2)
  (let ((p1 (eval xyz1))
      (p2 (eval xyz2)))
    (draw-line-xy *map* (first p1) (second p1)
                   (first p2) (second p2)
                   :width 2)))

(defun draw-dotted (edge)
  (let ((p1 (first edge))
      (p2 (second edge)))
    (draw-line-xy *map* (first p1) (second p1)
                   (first p2) (second p2)
                :dashing 2
                   :width 2)))

(defun draw-start (s)
    (draw-filled-circle-xy *map* (first  s)
                              (second s)
                          3))

(defun draw-goal (g)
    (draw-circle-xy *map*        (first g)
                              (second g)
                          3))

(defun draw-path-dots (current)
  (loop
    (cond ((null current)
         (return))
        (t
              (draw-filled-circle-xy *map* (first   (segment-exit-
point current))
                                                  (second  (segment-exit-
point current)) 2)))
             (setf current (segment-next-win current)))))

(defun draw-path-lines (current)
  (loop
    (cond ((null current)
         (return))
        (t
             (draw-line-xy *map*
                    (first (segment-exit-point
                                   (segment-prior-win current)))
                    (second (segment-exit-point
                                    (segment-prior-win current)))
                    (first  (segment-exit-point current))
                           (second (segment-exit-point current))
                    :dashing 3)))
             (setf current (segment-next-win current)))))

(defun draw-final (current)
  (let ((path-cost 0) (heading))
  (setf *bad-segments* nil)
  (loop
    (cond ((null current)
         (return))
        (t
         (unless (= (segment-cost current) 0.0)
;             (pprint "----------------")
;             (pprint "region")
;          (pprint (segment-region current))
           (setf heading  (rad-to-deg (angle
                        (cons (segment-exit-point
                                  (segment-prior-win current))
                                   (list
                                  (segment-exit-point current))))))
              (setf (segment-heading current) heading)
;                (pprint "segment heading")
;                (pprint heading)
;              (pprint "segment-cost")
;              (pprint (segment-cost current))
;                (cond ((illegal-heading current)
               (cond ((> (segment-cost current)
                    *infinity*)
                 (setf *bad-segments*
                 (append *bad-segments*
                      (list (segment-region current))))))
               (setf path-cost
               (+ path-cost (segment-cost current)))
               (draw-line-xy *map* (first  (segment-exit-point
                               (segment-prior-win current)))
                                   (second (segment-exit-point
                               (segment-prior-win current)))
                            (first  (segment-exit-point current))
                                        (second (segment-exit-point
current))
                        :dashing 3)
               (draw-ranges (segment-exit-point
                    (segment-prior-win current)) current))))
               (if (null (segment-next-win current))
              (draw-ranges *goal* current))
               (setf current (segment-next-win current)))
  (pprint "path-cost =")
  (pprint path-cost)
  (cond ((null *bad-segments*)
       (pprint "valid headings for all segments"))
      (t
         (pprint "impermissible path segments at:")
         (pprint *bad-segments*)))))

(defun draw-best-path (path)
  (let* ((path-cost (first path))
      (path-segments (rest path))
      (prior-point *start*)
      (current (first path-segments)))
  (clear *map*)
  (draw-regions *global-region-list*)
  (draw-filled-circle-xy *map* (first *start*)
                        (second *start*)
                        3)
  (loop
    (if (null current)
         (return))
    (unless (= (path-cost current) 0.0)
               (pprint "----------------")
               (pprint "region")
             (pprint (path-region current))
```

```
          (pprint "segment heading")
            (pprint (path-heading current))
        (pprint "segment-cost")
        (pprint (path-cost current))
           (draw-line-xy *map* (first  (path-point current))
                                  (second (path-point current))
                       (first  prior-point)
                                 (second prior-point)
                     :dashing 4)
        (draw-circle-xy *map* (first (path-point current))
                            (second (path-point current))
                            3)) ; end unless
        (setf prior-point (path-point current))
          (setf path-segments (rest path-segments))
        (setf current (first path-segments))) ; end loop
  (pprint "----------")
  (pprint "best path cost:")
  (pprint path-cost)))

(defun draw-direct (p1 p2 current)
    (draw-line-xy *map* (first p1) (second p1)
                  (first p2) (second p2)
               :dashing 3)
    (draw-ranges p1 *first-win*)
    (draw-ranges p2 *first-win*))

(defun draw-bounds (element)
    (loop
      (unless (or (null element)
            (= (segment-cost element) 0.0))
        (draw-ranges (segment-exit-point
                (segment-prior-win element)) element)
        (if (null (segment-next-win element))
          (draw-ranges (segment-exit-point
                  (segment-prior-win current)) element)
        (setf element (segment-next-win element))
        (if (null element)
          (return))))))

(defun draw-circle (current)
  (draw-circle-xy *map* (first  (segment-exit-point current))
                      (second (segment-exit-point current)) 2))

(defun draw-dot (current)
   (draw-filled-circle-xy  *map*  (first   (segment-exit-point
current))
                            (second (segment-exit-point current)) 2))

(defun draw-d (point)
  (draw-filled-circle-xy *map* (first point)
                            (second point) 2))
(defun whiteout (current)
  (let ((p1 (segment-exit-point (segment-prior-win current)))
      (p2 (segment-exit-point current))
      (p3 (segment-exit-point (segment-next-win current))))
    (setf (window-stream-foreground-color *map*) white)
    (draw-line-xy *map* (first p1) (second p1)
                        (first p2) (second p2))
    (draw-line-xy *map* (first p2) (second p2)
                        (first p3) (second p3))
    (setf (window-stream-foreground-color *map*) black)))

(defun draw-current (element)
  (unless (null (segment-prior-win element))
    (draw-line-xy *map* (first  (segment-exit-point (segment-prior-
win element)))
                        (second (segment-exit-point (segment-prior-win
element)))
                        (first  (segment-exit-point element))
                      (second (segment-exit-point element))
                      :dashing 4))
  (unless (null (segment-next-win element))
    (draw-line-xy *map* (first  (segment-exit-point element))
                        (second (segment-exit-point element))
                         (first   (segment-exit-point (segment-next-win
element)))
                      (second   (segment-exit-point  (segment-next-win
element)))
                      :dashing 4)))

(defun draw-r (p1 p2)
  (unless (or (null p1)
              (null (first p2)))
  (draw-line-xy *map* (first p1) (second p1)
                      (first (first p2)) (second (first p2)))))

(defun qdraw ()
  (clear *map*)
  (draw-start *start*)
  (draw-goal *goal*)
  (draw-regions *global-region-list*)
  (draw-path-lines *first-win*))

(defun draw-r (p1 p2)
  (unless (or (null p1)
              (null (first p2)))
  (draw-line-xy *map* (first p1) (second p1)
                      (first (first p2)) (second (first p2)))))
```

```
;*****************************************************************
;
;  Filename:  geometry.cl
;
;  By:  HILTON, Cary A.
;
;  Date:  May 91
;
;  Function:  contains geometric spatial
;             reasoning functions
;
;              1.  includes functions to determine angles of line
segments,
;                   and critical angles for power, sideslope, and
braking
;                  limits
;             2.  determines if a vehicle heading is inside
;                 a non-permissible heading range
;             3.  computes line intersections
;
;*****************************************************************


(setf *ray-length* 920)    ; extend ray to intersect edge
(setf *radius*  12)        ; radius for sideslope range icon
(setf *radius2* 8)         ; radius for power range icon
(setf *offset* (/ pi 2))   ; orient north to 0 degrees

(defun angle (edge)
  (let ((p1 (first  edge))
      (p2 (second edge))
      (offset (/ pi 2))
      (angle))
  (if (or (eq p1 nil) (eq p2 nil))
      (setf angle nil)
    (setf angle (- offset (atan (- (second p2) (second p1))
                  (- (first  p2) (first  p1))))))
    (mod angle *pi2*)))

(defun angle-between (e1 e2)
  (sin (abs (- (angle e1) (angle e2)))))

(defun deg-to-rad (degrees)
  (* degrees 0.0174533))

(defun rad-to-deg (radians)
  (/ radians 0.0174533))

;  accepts radian measure
(defun heading-p-b (slope limit)
  (cond ((and (> slope 0)
            (> slope (abs limit)))
       (acos (/ (tan limit) (tan slope))))
      (t 0)))
```

```
;  accepts radian measure
(defun heading-s (slope limit)
  (cond ((and (> slope 0)
              (> slope limit))
       (asin (/ (tan limit) (tan slope))))
      (t 0)))

(defun calc-ray (point angle)
  (let* ((x (+ (* *ray-length* (sin angle))  (first point)))
         (y (+ (* *ray-length* (cos angle)) (second point)))
         (end-pt (cons x (list y))))
   (cons point (list end-pt))))

(defun find-region (point)
  (let* ((region-list *global-region-list*)
       (current-region (first region-list))
       (inside))
    (loop
      (cond ((null region-list)
           (return))
          (t
           (setf inside (inside-region point current-region))
           (if inside
             (return))
           (setf region-list (rest region-list))
           (setf current-region (first region-list)))))
    inside))

(defun betweenp (a b x)
  (or (and (>= x a) (<= x b))
      (and (>= x b) (<= x a))))

(defun on-segment (edge point)
  (let ((x1 (first (first edge)))
      (y1 (second (first edge)))
      (x2 (first (second edge)))
      (y2 (second (second edge)))
      (xt (first point))
      (yt (second point)))
    (and (and (betweenp x1 x2 xt)
            (betweenp y1 y2 yt))
       (zerop (- (* (- x2 x1) (- yt y1))
                (* (- y2 y1) (- xt x1)))))))

(defun inside-region (point region)
  (let* ((vlist (polygon-vlist (eval region)))
       (ray (calc-ray point pi))
       (edge (cons (eval (first (last vlist)))
                (list (eval (first vlist)))))
       (intercept (find-int ray edge))
       (cross-count 0))
    (unless (null intercept)
      (setf intercept (cons (float (first  intercept))
                      (list (float (second intercept))))))
    (if (cross-p ray edge)
      (incf cross-count))
      (if (equal intercept (butlast (first edge)  1))
          (incf cross-count))
      (loop
      (setf edge (cons (eval (first vlist))
                    (list (eval (second vlist)))))
      (if (cross-p ray edge)
          (incf cross-count))
      (if (equal intercept (butlast (first edge)  1))
          (incf cross-count))
      (setf vlist (rest vlist))
      (if (null (rest vlist))
          (return)))
      (if (oddp cross-count)
         region
        nil)))

;  accepts radian measure
(defun draw-ranges (point current)
  (let
    ((p-x1) (p-y1) (p-x2) (p-y2)
     (r-x1) (r-y1) (r-x2) (r-y2)
     (l-x3) (l-y3) (l-x4) (l-y4)
     (b-x1) (b-y1) (b-x2) (b-y2)
     (slope (segment-slope current)))
;-----------------------------------------------------------
    (unless (null (segment-critical-p current))
         (setf  p-x1  (+  (*  *radius2*  (sin  (segment-critical-p
current)))  (first point)))
         (setf  p-y1  (+  (*  *radius2*  (cos  (segment-critical-p
current))) (second point)))
       (setf p-x2 (+ (* *radius2* (sin (segment-dual-p current)))
(first point)))
       (setf p-y2 (+ (* *radius2* (cos (segment-dual-p current)))
(second point))))
;-----------------------------------------------------------
    (unless (null (segment-critical-r current))
    (setf r-x1 (+ (* *radius* (sin (segment-critical-r current)))
(first point)))
    (setf r-y1 (+ (* *radius* (cos (segment-critical-r current)))
(second point)))
     (setf  r-x2  (+  (*  *radius*  (sin  (segment-dual-r  current)))
(first point)))
     (setf  r-y2  (+  (*  *radius*  (cos  (segment-dual-r  current)))
(second point))))
;-----------------------------------------------------------
    (unless (null (segment-critical-l current))
      (setf l-x3 (+ (* *radius* (sin (segment-critical-l current)))
(first point)))
      (setf l-y3 (+ (* *radius* (cos (segment-critical-l current)))
(second point)))
       (setf  l-x4  (+  (*  *radius*  (sin  (segment-dual-l  current)))
(first point)))
       (setf  l-y4  (+  (*  *radius*  (cos  (segment-dual-l  current)))
(second point))))
```

```
;-------------------------------------------------------------
    (unless (null (segment-critical-b current))
      (setf b-x1 (+ (* *radius* (sin (segment-critical-b current)))
(first point)))
      (setf b-y1 (+ (* *radius* (cos (segment-critical-b current)))
(second point)))
       (setf b-x2 (+ (* *radius* (sin (segment-dual-b current)))
(first point)))
       (setf b-y2 (+ (* *radius* (cos (segment-dual-b current)))
(second point))))
;-------------------------------------------------------------

  (draw-circle-xy *map*
                 (first point)
                 (second point)
                 *radius*
                 :dashing 5)
  (cond ((> slope 0)
       (if (> slope (vehicle-angle-p *vehicle*))
           (draw-filled-triangle-xy *map*
                                    (first point)
                                    (second point)
                                   p-x1
                                   p-y1
                                   p-x2
                                   p-y2)
         nil)
       (cond ((> slope (vehicle-angle-s *vehicle*))
            (draw-filled-triangle-xy *map*
                                       (first point)
                                       (second point)
                                      r-x1
                                      r-y1
                                      r-x2
                                      r-y2)
            (draw-filled-triangle-xy *map*
                                       (first point)
                                       (second point)
                                      l-x3
                                      l-y3
                                      l-x4
                                      l-y4))
             (t nil))
             (if (> slope (abs (vehicle-angle-b *vehicle*)))
               (draw-triangle-xy *map*
                                  (first point)
                                  (second point)
                                 b-x1
                                 b-y1
                                 b-x2
                                 b-y2
                                 :dashing 3)
            nil))
      (t nil))))
```

```
(defun sign (x)
  (cond ((plusp x) 1)
      ((minusp x) -1)
        ((zerop x) 0)))

(defun area (p1 p2 p3)
  (- (* (- (first p2) (first p1)) (- (second p3) (second p1)))
     (* (- (first p3) (first p1)) (- (second p2) (second p1)))))

(defun slope-and-int (p1 p2)
  (let ((x1 (first p1))
      (y1 (second p1))
      (x2 (first p2))
      (y2 (second p2)))
    (cond ((= x1 x2) (list nil x1))
        (t (list (/ (- y2 y1) (- x2 x1))
               (- y2 (* x2 (/ (- y2 y1) (- x2 x1)))))))))

(defun find-pt (mb1 mb2)
  (let ((m1 (first mb1))
      (b1 (second mb1))
      (m2 (first mb2))
      (b2 (second mb2)))
    (cond ((equal m1 m2)
         (cond ((not (= b1 b2)) (list nil nil))
             (t (list nil b1))))
        ((eq m1 nil) (list b1 (+ b2 (* m2 b1))))
        ((eq m2 nil) (list b2 (+ b1 (* m1 b2))))
        (t (list (fround (/ (- b2 b1) (- m1 m2)))
              (fround (+ b1 (* m1 (/ (- b2 b1) (- m1 m2))))))))))

(defun cross-p (e1 e2)
  (let ((p1 (first  e1))
      (p2 (second e1))
      (p3 (first  e2))
      (p4 (second e2)))
  (if (and (/= (sign (area p1 p2 p3))
               (sign (area p1 p2 p4)))
           (/= (sign (area p3 p4 p1))
               (sign (area p3 p4 p2))))
        t
        nil)))

;  Find intersection of two segments

(defun find-int (e1 e2)
  (let ((mb-one) (mb-two) (int))
    (cond ((cross-p e1 e2)
           (setf mb-one (slope-and-int (first e1) (second e1)))
         (setf mb-two (slope-and-int (first e2) (second e2)))
           (setf int (find-pt mb-one mb-two)))
        (t nil))))

;  Finds intersection of two lines
```

```
(defun line-int (e1 e2)
  (let ((mb-one (slope-and-int (first e1) (second e1)))
       (mb-two (slope-and-int (first e2) (second e2))))
          (find-pt mb-one mb-two)))

(defun find-valid-crossing (point current)
  (let ((ray-t) (valid-point))
    (setf ray-t (calc-ray point (segment-critical-r current)))
    (setf valid-point (find-int ray-t (segment-frontier current)))
    (if (null valid-point)
      (setf ray-t (calc-ray point (segment-dual-r current)))
         (setf  valid-point  (find-int  ray-t  (segment-frontier
current))))
    (if (null valid-point)
      (setf ray-t (calc-ray point (segment-critical-l current)))
         (setf  valid-point  (find-int  ray-t  (segment-frontier
current))))
    (if (null valid-point)
      (setf ray-t (calc-ray point (segment-dual-l current)))
         (setf  valid-point  (find-int  ray-t  (segment-frontier
current))))
    valid-point))



(defun mark-pt (int)
  (draw-filled-circle-xy *win1* (first int) (second int) 5))

(defun draw-line (line)
  (draw-line-xy *win1* (first (first line))  (second (first line))
                 (first (second line)) (second (second line))
                 :brush-width 2))

(defun find-elev (point current)
  (let* ((region (eval current))
        (crossing-pts (find-elev-points point region)))
    (calc-elev point crossing-pts)))


(defun find-elev-points (point current)
  (let ((vlist (polygon-vlist current))
       (edge) (int1) (int2)
          (elev1) (elev2)
       (ray1 (calc-ray point pi))
       (ray2 (calc-ray point (* pi 2))))
    (setf edge (cons (eval (first vlist))
                (list (eval (first (last vlist))))))
    (setf int1 (find-int ray1 edge))
    (if int1
      (setf elev1 (calc-elev int1 edge)))
    (setf int2 (find-int ray2 edge))
    (if int2
      (setf elev2 (calc-elev int2 edge)))
    (loop
      (setf edge (cons (eval (first vlist))
                    (list (eval (second vlist)))))
          (cond ((null int1)
                 (setf int1 (find-int ray1 edge))
                   (if int1
                    (setf elev1 (calc-elev int1 edge)))))
          (cond ((null int2)
                 (setf int2 (find-int ray2 edge))
                   (if int2
                    (setf elev2 (calc-elev int2 edge)))))
          (setf vlist (rest vlist))
          (if (or (and int1 int2)
                  (null (rest vlist)))
            (return)))
        (setf int1 (append int1 (list elev1)))
        (setf int2 (append int2 (list elev2)))
        (cons int1 (list int2))))

(defun calc-elev (point edge)
  (let* ((end-pt1 (first edge))
         (end-pt2 (second edge))
         (edge-d) (point-d)
           (low (min (third end-pt1) (third end-pt2)))
         (elev-d (abs (- (third end-pt1)
                        (third end-pt2)))))
    (cond ((= (first end-pt1)
              (first end-pt2))
           (setf edge-d  (abs (- (second end-pt1)
                                 (second end-pt2))))
           (setf point-d (abs (- (second end-pt1)
                                 (second point)))))
          (t
           (setf edge-d  (abs (- (first end-pt1) (first end-pt2))))
           (setf point-d (abs (- (first end-pt1) (first point))))))
    (if (zerop elev-d)
      (third end-pt1)
    (+ (/ (* point-d elev-d) edge-d) low))))

(defun calc-displ (p1 p2)
  (let ((x1 (first p1))
        (y1 (second p1))
        (x2 (first p2))
        (y2 (second p2)))
    (+ (abs (- x1 x2))
       (abs (- y1 y2)))))

(defun distance (p1 p2)
  (let*  ((x1 (first p1))
          (y1 (second p1))
          (x2 (first p2))
        (y2 (second p2))
          (x  (- x2 x1))
          (y  (- y2 y1)))
    (sqrt (+ (* x x) (* y y)))))

(defun w-distance (p1 p2 w)
  (let*  ((x1 (first p1))
          (y1 (second p1))
          (x2 (first p2))
        (y2 (second p2))
          (x  (- x2 x1))
          (y  (- y2 y1)))
    (* (sqrt (+ (* x x) (* y y))) w)))

(defun find-edge (r1 r2)
  (let (edge)
    (dolist (element (polygon-vlist r1) edge)
      (if (memberp element (polygon-vlist r2))
         (setf edge (append edge (list element)))))
    (cond ((null (rest edge))
        (append edge edge))
          ((and (equal (first edge)
                  (first (polygon-vlist r1)))
              (equal (second edge)
                  (first (last (polygon-vlist r1)))))
        (reverse edge))
        (t
          edge))))

(defun mid-point (p1 p2)
  (let ((x1 (first  p1))
      (y1 (second p1))
      (x2 (first  p2))
      (y2 (second p2)))
    (cond ((or (eq nil x1)
             (eq nil x2)) (list nil nil))
        (t
         (list (* 0.5 (+ x1 x2)) (* 0.5 (+ y1 y2)))))))

(defun illegal-heading (current)
  (let ((angle (angle (cons
                  (segment-exit-point (segment-prior-win current))
                  (list (segment-exit-point current)))))
      (cr (segment-critical-r current))
      (dr (segment-dual-r current))
      (cl (segment-critical-l current))
      (dl (segment-dual-l current)))
  (if (or (inside-pl angle cl dl)
        (inside-rb angle cr dr))
      t
    nil)))

(defun inside-pl (angle critical dual)
  (cond ((> critical dual)
       (if (and (< angle (- critical .0174))
              (> angle (+ dual .0174)))
           t
          nil))
      ((> dual critical)
         (if (and (> angle (- critical .0174))
               (< angle (+ dual .0174)))
              nil
            t))
      (t nil)))

(defun inside-rb (angle critical dual)
  (cond ((> critical dual)
       (if (and (> angle (- dual .0174))
             (< angle (+ critical .0174)))
           nil
         t))
      ((> dual critical)
         (if (and (> angle (+ critical .0174))
               (< angle (- dual .0174)))
             t
           nil))
      (t nil)))

(defun virtual-obstacle (current)
  (let* ((region (eval current))
       (ps-rad (deg-to-rad (polygon-slope region)))
       (vp-rad (deg-to-rad (vehicle-angle-p *vehicle*)))
       (vs-rad (deg-to-rad (vehicle-angle-s *vehicle*)))
       (critical-p (heading-p-b ps-rad vp-rad))
       (critical-s (heading-s   ps-rad vs-rad)))
    (if (< critical-p critical-s)
     t
     nil)))

(defun valid-entry (edge-from edge-to region)
  (let* ((r (eval region))
       (pf1) (pf2) (pt1) (pt2)
       (rotation)
       (slope (deg-to-rad (polygon-slope r)))
       (cr) (dr) (cl) (dl)
       (h1) (h2) (hdg1) (hdg2))
    (cond ((> slope
            (vehicle-angle-s *vehicle*))
         (setf pf1 (first edge-from))
         (setf pf2 (second edge-from))
         (setf pt1 (first edge-to))
         (setf pt2 (second edge-to))
         (setf rotation (deg-to-rad (polygon-orientation r)))
         (setf cr (heading-s slope
                         (vehicle-angle-s *vehicle*)))
         (setf dr (mod (+ (- pi cr) rotation) *pi2*))
         (setf cl (mod (+ (- cr) rotation) *pi2*))
         (setf dl (mod (+ (- cr pi) rotation) *pi2*))
           (setf h1 (cons pf1 (list pt2)))
         (setf h2 (cons pf2 (list pt1)))
         (setf hdg1 (angle h1))
         (setf hdg2 (angle h2))
           (setf cr (+ cr rotation))
           (if (or (and (inside-rb hdg1 cr dr)
                   (inside-rb hdg2 cr dr))
                 (and (inside-pl hdg1 cl dl)
                  (inside-pl hdg2 cl dl)))
         nil
         t))
     (t
      t))))
```

```
;*****************************************************************
;
;  Filename:  links2.cl
;
;  By:  HILTON, Cary A.
;
;  Date:  May 91
;
;  Function:  Processes region list into window list
;
;             1.  converts list into doubly linked
;                 circular list
;             2.  creates 'segment' structure for use in
;                 evaluating window sequences.
;             3.  computes critical headings,
;                 initial crossing points,
;                 edge common to adjacent regions
;
;*****************************************************************


(setf *init*    0)        ;  initialize edge count:  odd points to
left, even points to right
(setf *print-circle* 1)  ;  turn off print loop for circular data
structures (doubly linked lists)

(defconstant *pi2* (* pi 2))

;****************************************************************

(defstruct segment
  index
  region
  exit-point
  heading
  prior-point
  displ
  frontier
  frontier-length
  weight
  cost
  next-win
  prior-win
  critical-p
  dual-p
  critical-r
  dual-r
  critical-l
  dual-l
  critical-b
  dual-b
  slope
  orientation)

(defun memberp (item input-list)
  (if input-list
      (if (eq item
           (first input-list))
             t
           (memberp item (rest input-list)))))


(defun double-link (head tail)
    (unless (null tail)
      (setf (segment-next-win head) (first tail))
      (setf (segment-prior-win (first tail)) head)
      (double-link (first tail) (rest tail))))

(defun change-struct (start goal rlist)
  (let* ((win-t) (weight-t)
        (exit-t) (aft-t) (next-t)
        (verts) (edge-t) (hdg-t) (ray-t)
        (r) (r-next)
        (pt1) (pt2) (ecount 0)
        (pwr-rad) (side-rad) (brk-rad)
        (crit-p) (crit-r) (crit-l) (crit-b)
        (cp) (dp) (cr) (dr)
        (cl) (dl) (cb) (db)
        (slope-rad) (oriented))
    (if rlist
       (setf win-t (list (make-segment        :index       0
                                         :region      (first rlist)
                                              :exit-point  start
                                                     :frontier    (cons
                                                 start
                                                 (list start))))))
    (loop
      (cond ((null rlist)
           (return))
          (t
           (setf r (eval (first rlist)))
           (setf r-next (eval (second rlist)))
           (incf ecount)
             (cond (r-next
                    (setf verts (find-edge r r-next))
                    (setf pt1 (eval (first  verts)))
                    (setf pt2 (eval (second verts))))
                 (t
                  (setf pt1 goal)
                  (setf pt2 goal)))))
             (setf edge-t (cons pt1 (list pt2)))
             (setf weight-t (polygon-weight r))
             (setf slope-rad (deg-to-rad (polygon-slope r)))
             (setf oriented  (deg-to-rad (polygon-orientation r)))
             (setf pwr-rad   (vehicle-angle-p *vehicle*))
             (setf side-rad  (vehicle-angle-s *vehicle*))
             (setf brk-rad   (vehicle-angle-b *vehicle*))
               (setf crit-p (heading-p-b slope-rad pwr-rad))
               (setf crit-r (heading-s slope-rad side-rad))
             (setf crit-l (- crit-r))
```

```
            (setf crit-b (heading-p-b slope-rad brk-rad))
            (setf cp (mod (+ crit-p oriented) *pi2*))
            (setf dp (mod (+ (- crit-p) oriented) *pi2*))
            (setf cr (mod (+ crit-r oriented) *pi2*))
            (if (= crit-r crit-l)
              (setf dr (mod (+ crit-r oriented) *pi2*))
            (setf dr (mod (+ (- pi crit-r) oriented) *pi2*)))
            (setf cl (mod (+ crit-l oriented) *pi2*))
            (if (= crit-r crit-l)
              (setf dl (mod (+ crit-l oriented) *pi2*))
            (setf dl (mod (+ (- crit-r pi) oriented) *pi2*)))
            (setf cb (mod (+ crit-b oriented) *pi2*))
            (setf db (mod (+ (- crit-b) oriented) *pi2*))
            (cond ((inside-rb cb cr dr)
                (setf cb dr)
                (setf db dl)))
;             (cond ((oddp ecount)                  ; set point at
frontier endpoint
;               (setf exit-t (first edge-t)))
;              ((evenp ecount)
;               (setf exit-t   (second edge-t))))
            (setf exit-t (mid-point pt1 pt2)        ; set point at
                                                    ; frontier
midpoint

            (setf temp (make-segment  :index         ecount
                         :region        (first rlist)
                         :exit-point    exit-t
                                    :prior-point   exit-t
                         :displ         (+ *point-tolerance* 1)
                                    :frontier      edge-t
                                    :frontier-length
                                    (distance (first  edge-t)
                                  (second edge-t))
                         :weight        weight-t
                                    :cost          nil
                         :next-win      nil
                         :prior-win     nil
                         :slope         slope-rad
                         :orientation   oriented
                         :critical-p    cp
                         :dual-p        dp
                         :critical-r    cr
                         :dual-r        dr
                         :critical-l    cl
                         :dual-l        dl
                         :critical-b    cb
                         :dual-b        db))
              (setf win-t (append win-t (list temp)))
            (setf rlist (rest rlist))
            (if (null rlist)
              (return win-t)))))
```

```
;*****************************************************************
;
;  Filename:  partition.cl
;
;  By:  HILTON, Cary A.
;
;  Date:  May 91
;
;  Function:  These functions partition the
;              current window edge at the
;              points where the cost function
;              changes.
;
;              1.  The possible cost functions are:
;                  pp, pi, pb, ip, ii, ib
;                  bp, bu, bb.
;                  "p" = power range
;                  "i" = isotropic (unrestricted)
;                  "b" = braking range
;
;              2.  performs golden ratio
;                  search with appropriate cost functions.
;
;*****************************************************************

(defvar *infinity* 65536) ; represents cost of "infinity"

(defvar *proot*)          ;  root for list of transition
                          ;   points along a frontier

(defstruct partition
  displ
  t-point
  frontier
  range
  next
  prior)

(defun insert-part (new root)
  (let ((top root) (current root))
    (cond ((null current)
           (setf current new))
          ((< (partition-displ new) (partition-displ current))
           (setf (partition-prior current) new)
           (setf (partition-next new) current)
           (setf current new))
          (t  (loop
              (if  (and  (>=  (partition-displ  new)  (partition-displ
current))
                         (not (null (partition-next current))))
                     (setf current (partition-next current))
                (return)))
                  (cond  ((< (partition-displ  new)  (partition-displ
current))
                         (setf (partition-next new) current)
```

```
                    (setf (partition-prior new) (partition-prior
current))
                  (setf (partition-next (partition-prior current))
new)
                (setf (partition-prior current) new)
                (setf current top))
                (t
                 (setf (partition-next new) nil)
                    (setf (partition-next current) new)
                    (setf (partition-prior new) current)
                  (setf current top)))))))


(defun insert-plist (plist)
  (let (root)
  (dolist (element plist root)
    (setf root (insert-part element root)))))

(defun pre-part (current)
  (let* ((point (segment-exit-point (segment-prior-win current)))
       (edge (segment-frontier current))
       (rcp) (rdp) (rcr) (rdr)
       (rcl) (rdl) (rcb) (rdb)
       (cp-int) (dp-int) (cr-int) (dr-int)
       (cl-int) (dl-int) (cb-int) (db-int)
         (plist))
     (cond ((or (eq point (first edge))
             (eq point (second edge)))
          nil)
         (t
     (unless (= (segment-critical-p current)
             (segment-dual-p current))
         (setf rcp (calc-ray point (segment-critical-p current)))
       (setf rdp (calc-ray point (segment-dual-p current)))
         (setf cp-int (cons (find-int rcp edge) '(i)))
       (setf dp-int (cons (find-int rdp edge) '(p))))
     (unless (= (segment-critical-r current)
             (segment-dual-r current))
       (setf rcr (calc-ray point (segment-critical-r current)))
       (setf rdr (calc-ray point (segment-dual-r current)))
         (setf rcl (calc-ray point (segment-critical-l current)))
       (setf rdl (calc-ray point (segment-dual-l current)))
         (setf cr-int (cons (find-int rcr edge) '(s)))
       (setf dr-int (cons (find-int rdr edge) '(i)))
         (setf cl-int (cons (find-int rcl edge) '(i)))
       (setf dl-int (cons (find-int rdl edge) '(s))))
     (unless (= (segment-critical-b current)
             (segment-dual-b current))
       (setf rcb (calc-ray point (segment-critical-b current)))
       (setf rdb (calc-ray point (segment-dual-b current)))
         (setf cb-int (cons (find-int rcb edge) '(b)))
       (setf db-int (cons (find-int rdb edge) '(i))))
    (append (list dp-int) (list cr-int) (list dr-int) (list cb-int)
          (list db-int) (list dl-int) (list cl-int) (list dp-
int))))))

(defun post-part (current)
  (let* ((point (segment-exit-point (segment-next-win current)))
        (edge (segment-frontier current))
        (rcp) (rdp) (rcr) (rdr)
        (rcl) (rdl) (rcb) (rdb)
        (cp-int) (dp-int) (cr-int) (dr-int)
        (cl-int) (dl-int) (cb-int) (db-int)
          (plist))
(cond ((or (eq point (first edge))
          (eq point (second edge)))
        nil)
       (t

      (unless (= (segment-critical-p (segment-next-win current))
           (segment-dual-p (segment-next-win current)))
          (setf rcp (calc-ray point (+ pi (segment-critical-p
                                  (segment-next-win current)))))
        (setf rdp (calc-ray point (+ pi (segment-dual-p
                                  (segment-next-win current)))))
          (setf cp-int (cons (find-int rcp edge) '(p)))
        (setf dp-int (cons (find-int rdp edge) '(i))))
      (unless (= (segment-critical-r (segment-next-win current))
           (segment-dual-r (segment-next-win current)))
        (setf rcr (calc-ray point (+ pi (segment-critical-r
                                  (segment-next-win current)))))
        (setf rdr (calc-ray point (+ pi (segment-dual-r
                                  (segment-next-win current)))))
          (setf rcl (calc-ray point (+ pi (segment-critical-l
                                  (segment-next-win current)))))
        (setf rdl (calc-ray point (+ pi (segment-dual-l
                                  (segment-next-win current)))))
          (setf cr-int (cons (find-int rcr edge) '(i)))
        (setf dr-int (cons (find-int rdr edge) '(s)))
          (setf cl-int (cons (find-int rcl edge) '(s)))
        (setf dl-int (cons (find-int rdl edge) '(i))))
      (unless (= (segment-critical-b (segment-next-win current))
           (segment-dual-b (segment-next-win current)))
        (setf rcb (calc-ray point (+ pi (segment-critical-b
                                  (segment-next-win current)))))
        (setf rdb (calc-ray point (+ pi (segment-dual-b
                                  (segment-next-win current)))))
          (setf cb-int (cons (find-int rcb edge) '(i)))
        (setf db-int (cons (find-int rdb edge) '(b))))
     (append (list cp-int) (list cr-int) (list dr-int) (list cb-int)
          (list db-int) (list dl-int) (list cl-int) (list dp-
int))))))

(defun part (r1 r2)
  (let*  ((frontier (segment-frontier r1))
         (pre-point
           (segment-exit-point (segment-prior-win r1)))
         (t-point
          (first (segment-frontier r1)))
        (post-point
          (segment-exit-point r2))
        (pre-trans-pts  (pre-part r1))
        (post-trans-pts (post-part r1))
        (init-range)
        (t-list)
        (segment-t))
    (setf t-list
      (dolist (element pre-trans-pts t-list)
                (unless (null (first element))
                  (setf temp (make-partition
                               :displ    (calc-displ (first element) t-
point)
                              :t-point  (first element)
                              :frontier 'pre
                              :range    (second element)))
                  (setf t-list (append t-list (list temp))))))
    (setf t-list
      (dolist (element post-trans-pts t-list)
                (unless (null (first element))
                  (setf temp (make-partition
                               :displ    (calc-displ (first element) t-
point)
                              :t-point  (first element)
                              :frontier 'post
                              :range    (second element)))
                  (setf t-list (append t-list (list temp))))))
    (setf temp (make-partition
                :displ   *infinity*
           :t-point (second frontier)
             :frontier nil
           :range    nil))
    (setf t-list (append t-list (list temp)))
    (setf *proot* (insert-plist t-list))))


(defun adjust-crossing (current)
  (let* ((r1 current) (r2 (segment-next-win current))
       (pre-point
          (segment-exit-point (segment-prior-win r1)))
         (t-point (first (segment-frontier r1)))
       (post-point (segment-exit-point r2))
       (pre-segment  (cons pre-point (list t-point)))
       (post-segment (cons t-point (list post-point)))
       (best-frontier) (frontier-t) (best-range)
       (end-pt1) (end-pt2) (ray-t)
       (pre-range) (post-range)
         (best-crossing)
       (next-crossing) (loop-flag))
    (setf pre-range (find-pre-range pre-segment r1))
    (setf post-range (find-post-range post-segment r2))
    (part r1 r2)
    (loop
      (setf end-pt1 t-point)
      (setf end-pt2 (partition-t-point *proot*))
      (setf frontier-t (cons end-pt1 (list end-pt2)))
      (setf next-crossing (costf pre-point pre-range
                             frontier-t
                             post-point post-range
                           current
                         nil))

      (cond ((or (null best-crossing)
               (< (first next-crossing) (first best-crossing)))
          (setf best-crossing next-crossing)
          (setf best-frontier frontier-t)
          (setf best-range (fifth best-crossing))))
      (if (equal (partition-frontier *proot*) 'pre)
        (setf pre-range (partition-range *proot*))
        (setf post-range (partition-range *proot*)))
      (setf *proot* (partition-next *proot*))
      (if (null *proot*)
        (return)
        (setf t-point (second frontier-t)))) ; end loop

     (setf best-crossing (costf pre-point
                         (first best-range)
                             best-frontier
                             post-point
                         (second best-range)
                           current
                         t))


    (cond ((< (first best-crossing)
           (+ (segment-cost r1)
            (segment-cost r2)))

            (setf (segment-prior-point r1) (segment-exit-point r1))
            (whiteout current)
            (draw-region (segment-region current))
            (setf (segment-cost r1) (second best-crossing))
            (setf (segment-cost r2)  (third  best-crossing))
            (setf (segment-exit-point r1) (fourth best-crossing))
            (draw-current current)
            (setf (segment-displ r1)
           (calc-displ (segment-prior-point r1)
                   (segment-exit-point  r1))))))) ; end cond, let
defun

(defun print-plist (current)
  (pprint "plist")
  (loop
    (pprint (partition-t-point current))
    (pprint (partition-range current))
    (pprint"---------------")
    (setf current (partition-next current))
    (if (null current)
      (return))))
```

```
(defun calc-pre-range (heading current)
  (cond ((inside-pl heading
              (segment-critical-p current)
              (segment-dual-p current))
        'p)
         ((inside-rb heading
              (segment-critical-r current)
              (segment-dual-r current))
        's)
         ((inside-pl heading
              (segment-critical-l current)
              (segment-dual-l current))
        's)
       ((inside-rb heading
              (segment-critical-b current)
              (segment-dual-b current))
        'b)
       (t 'i)))

(defun calc-post-range (heading current)
  (cond ((inside-pl heading
              (+ pi (segment-dual-p current))
              (+ pi (segment-critical-p current)))
        'p)
         ((inside-rb heading
              (+ pi (segment-dual-r current))
              (+ pi (segment-critical-r current)))
        's)
         ((inside-pl heading
              (+ pi (segment-dual-l current))
              (+ pi (segment-critical-l current)))
        's)
       ((inside-rb heading
              (+ pi (segment-dual-b current))
              (+ pi (segment-critical-b current)))
        'b)
       (t 'i)))

(defun find-post-range (segment region)
  (let ((hdg) (range))
       (if (eq (first segment)
              (second segment))
         (setf hdg
           (angle (segment-frontier
                 (segment-prior-win region))))
         (setf hdg (angle segment)))
       (setf range (calc-pre-range hdg region))))


(defun find-pre-range (segment region)
  (let ((hdg) (range))
     (if (eq (first segment)
           (second segment))
       (setf hdg
  (angle (segment-frontier region)))
(setf hdg (angle segment)))
(setf range (calc-pre-range hdg region))))
```

```
;*********************************************************************
;
;  Filename:  start.cl
;
;  By:  HILTON, Cary A.
;
;  Date:  May 91
;
;  Function:  A* search returns list of regions
;             connecting start and goal points.
;
;              1.  heuristic:  current cost + weighted distance to
goal
;
;             2.  attempts to screen path segments for
;                 impermissible headings
;
;********************************************************************

(defvar *nroot* nil)

;  This structure forms a linked list
;    ordered by cost which determines
;    the next node for expansion during
;    the A* region search.

(defstruct node
  point
  edge
  chain
  region
  cost
  cost-est
  prior
  next)

(defun region-search (start goal
                 start-region
                 goal-region)
  (let* ((active-region start-region)
       (prior-edge (cons start (list start)))
       (current-edge)
       (active-node) (active-node-cost)
       (est-to-goal) (cost-tot 0) (temp-node)
       (prior-node start) (prior-region) (prior-cost 0)
       (back-path (list start-region)) (rlist))
;(pprint "start point is in region:")
;(pprint *start*)(pprint start-region)
;(pprint "goal point is in region:")
;(pprint *goal*)(pprint goal-region)
    (setf *nroot* nil)
    (setf current-node (make-node
                    :point        start
                           :edge          nil
                    :region       start-region
                    :chain          nil
                             :cost           0
                       :cost-est       0
                           :prior          nil
                       :next           nil))

;(pprint "detour-region")(pprint detour-region)
     (loop
       (dolist (element (polygon-alist (eval active-region)))
       (unless (or (and (eq element prior-region)
                      (not (eq start-region goal-region)))
                   (virtual-obstacle element)
                         (and (memberp element (node-chain current-
node))
                      (not (eq start-region goal-region))))
         (setf current-edge (find-edge (eval active-region)
                                   (eval element)))
         (setf current-edge (cons (eval (first current-edge))
                              (list
                               (eval (second current-edge)))))
         (unless (or (null (first current-edge))
                     (null (valid-entry prior-edge current-edge
                                      active-region)))


           (setf active-node (mid-point (first current-edge)
                                  (second current-edge)))
           (setf active-node-cost
             (+ (w-distance prior-node active-node
               (polygon-weight (eval active-region)))
              prior-cost))

             (draw-dotted (cons active-node (list prior-node)))
           (setf est-to-goal (distance active-node goal))
           (setf cost-tot    (+ active-node-cost est-to-goal))
           (setf temp-node (make-node
                    :point          active-node
                             :edge           current-edge
                    :region         element
                    :chain          back-path
                             :cost           active-node-cost
                       :cost-est       cost-tot
                           :prior          nil
                       :next           nil))
            (setf *nroot* (insert-node temp-node *nroot*)))))   ;end
unless, unless, dolist

         (if (null *nroot*)
           (return)
            (setf current-node *nroot*))
         (setf back-path  (append (node-chain current-node)
                              (list (node-region current-node))))
       (setf prior-cost (node-cost current-node))
       (if (eq (node-region current-node) goal-region)
         (return))
```

```
      (setf prior-edge (node-edge current-node))
      (setf *nroot* (node-next *nroot*))
      (setf prior-region active-region)
      (setf prior-node (node-point current-node))
      (setf active-region (node-region current-node)))

      (draw-dotted (cons (node-point current-node)
                         (list goal)))
      (setf rlist (append (node-chain current-node)
                          (list (node-region current-node))))
      (if (eq (first (last rlist)) goal-region)
        rlist
      nil)))

;*******************************************************************
*****

(defun insert-node (new root)
  (let ((top root) (current root))
;(pprint "inside insert-node")
    (cond ((null current)
           (setf current new))
        ((< (node-cost-est new) (node-cost-est current))
         (setf (node-prior current) new)
         (setf (node-next new) current)
         (setf current new))
        (t  (loop
            (if  (and  (>=  (node-cost-est  new)  (node-cost-est
current))
                   (not (null (node-next current))))
                (setf current (node-next current))
              (return)))
                  (cond ((< (node-cost-est new) (node-cost-est
current))
                   (setf (node-next new) current)
                   (setf (node-prior new) (node-prior current))
                   (setf (node-next (node-prior current)) new)
                 (setf (node-prior current) new)
                 (setf current top))
                 (t
                  (setf (node-next new) nil)
                     (setf (node-next current) new)
                     (setf (node-prior new) current)
                  (setf current top)))))))
```

```
;*******************************************************************
;
;  Filename:  vehicle.cl
;
;  By:  HILTON, Cary A.
;
;  Date:  May 91
;
;  Function:  Vehicle model.
;
;             1.  vehicle is defined by its power, sideslope and
;                 braking limits.
;
;             2.  angles are measured in degrees but converted
;                 by the program to radian measures.
;
;             3.  default is vehicle-2
;
;             4.  to change to another vehicle, enter:
;                 (setf *vehicle* (vehicle-rad vehicle-x))
;
;
;******************************************************************

(defstruct vehicle
  angle-p
  angle-s
  angle-b)

(defvar vehicle-1 (make-vehicle  :angle-p  31.0
                                :angle-s  17.0
                         :angle-b -10.0))

(defvar vehicle-2 (make-vehicle  :angle-p  26.0
                      :angle-s  22.0
                      :angle-b -5.0))

(defvar vehicle-3 (make-vehicle  :angle-p  31.0
                                 :angle-s  17.0
                      :angle-b -5.0))

(defvar vehicle-4 (make-vehicle  :angle-p   7.8
                         :angle-s   9.0
                         :angle-b  -8.0))

(defun vehicle-rad (vehicle)
  (make-vehicle  :angle-p (deg-to-rad (vehicle-angle-p vehicle))
            :angle-s (deg-to-rad (vehicle-angle-s vehicle))
            :angle-b (deg-to-rad (vehicle-angle-b vehicle))))
```

```
;*******************************************************************
;
;  Filename:  map2.cl
;
;  By:  HILTON, Cary A.
;
;  Date:  May 91
;
;  Function:  Contains map data.
;
;             1.  defines vertices, edges, and polygons within
;                 the map boundaries
;             2.  specifies which vertices and edges each polygon
;                 is made from
;             3.  contains adjacency list for polygons
;
;*******************************************************************

;  This map closely resembles the map used
;   by Ron Ross.

; *****************************************************************


(defstruct edge
  pt1
  pt2)

(defstruct polygon
  vlist
  slope
  orientation
  weight
  alist)

;*******************************************************************

;  Global vertex list

(setf *global-vertex-list*  '(v1 v2 v3 v4 v5 v6 v7 v8 v9
                       v10 v11 v12 v13 v14 v15 v16 v17 v18 v19
                       v20 v21 v22 v23 v24 v25 v26 v27 v28 v29
                       v30 v31 v32 v33 v34 v35 v36 v37 v38 v39
                       v40 v41 v42 v43 v44 v45))

;*******************************************************************

;  Global polygon list

(setf *global-region-list*  '(r1 r2 r3 r4 r5 r6 r7 r8 r9
                       r10 r11 r12 r13 r14 r15 r16 r17 r18 r19
                       r20 r21 r22 r23 r24 r25 r26 r27 r28 r29
                       r30 r31 r32 r33 r34 r35 r36 r37 r38))
(setf gl *global-region-list*)
```

```
;*******************************************************************

(setf  v1 '(352.0  96.0 14.5))
(setf  v2 '(480.0 224.0 14.5))
(setf  v3 '(400.0 304.0 14.5))
(setf  v4 '(272.0 176.0 14.5))

(setf  v5 '(240.0 320.0  9.0))
(setf  v6 '(240.0 448.0  9.0))
(setf  v7 '(112.0 448.0  9.0))
(setf  v8 '(112.0 320.0  9.0))

(setf  v9 '(192.0 368.0 27.75))
(setf v10 '(160.0 368.0 27.75))
(setf v11 '(192.0 400.0 27.75))
(setf v12 '(160.0 400.0 27.75))

(setf v13 '(352.0 160.0 35.75))
(setf v14 '(416.0 224.0 35.75))
(setf v15 '(400.0 240.0 35.75))
(setf v16 '(336.0 176.0 35.75))

(setf v17 '(352.0 112.0 18.0))
(setf v18 '(464.0 224.0 18.0))
(setf v19 '(400.0 288.0 18.0))
(setf v20 '(288.0 176.0 18.0))

(setf v21 '(352.0  48.0  0.0))
(setf v22 '(528.0 224.0  0.0))
(setf v23 '(400.0 352.0  0.0))
(setf v24 '(224.0 176.0  0.0))

(setf v25 '(208.0 352.0 19.5))
(setf v26 '(208.0 416.0 19.5))
(setf v27 '(144.0 416.0 19.5))
(setf v28 '(144.0 352.0 19.5))

(setf v29 '(256.0 304.0  9.0))
(setf v30 '(256.0 464.0  9.0))
(setf v31 '( 96.0 464.0  9.0))
(setf v32 '( 96.0 304.0  9.0))

(setf v33 '(288.0 272.0  0.0))
(setf v34 '(288.0 496.0  0.0))
(setf v35 '( 64.0 496.0  0.0))
(setf v36 '( 64.0 272.0  0.0))

(setf v37 '(  0.0 790.0  0.0))
(setf v38 '(600.0 790.0  0.0))
(setf v39 '(600.0 496.0  0.0))
(setf v40 '(600.0 224.0  0.0))
(setf v41 '(600.0   0.0  0.0))
(setf v42 '(352.0   0.0  0.0))
(setf v43 '( 64.0   0.0  0.0))
(setf v44 '(  0.0   0.0  0.0))
```

```
(setf v45 '(  0.0 496.0  0.0))


(setf  r1  (make-polygon    :vlist        '(v3 v23 v22 v2)
                    :slope        23.1
                    :orientation  225
                    :weight       1
                    :alist       '(r32 r2 r7 r4)))


(setf  r2  (make-polygon    :vlist        '(v19 v3 v2 v18)
                    :slope        17.2
                    :orientation  225
                    :weight       1
                    :alist       '(r1 r3 r8 r5)))

(setf  r3  (make-polygon    :vlist        '(v15 v19 v18 v14)
                    :slope        27.6
                    :orientation  225
                    :weight       1
                    :alist       '(r13 r2 r6 r9)))


(setf  r4  (make-polygon    :vlist        '(v21 v1 v2 v22)
                    :slope        23.1
                    :orientation  315
                    :weight       1
                    :alist       '(r1 r38 r10 r5)))

(setf  r5  (make-polygon    :vlist        '(v1 v17 v18 v2)
                    :slope        17.2
                    :orientation  315
                    :weight       1
                    :alist       '(r4 r6 r2 r11)))

(setf  r6  (make-polygon    :vlist        '(v17 v13 v14 v18)
                    :slope        27.6
                    :orientation  315
                    :weight       1
                    :alist       '(r5 r13 r3 r12)))

(setf  r7  (make-polygon    :vlist        '(v24 v23 v3 v4)
                    :slope        23.1
                    :orientation  135
                    :weight       1
                    :alist       '(r34 r8 r1 r10)))

(setf  r8  (make-polygon    :vlist        '(v4 v3 v19 v20)
                    :slope        17.2
                    :orientation  135
                    :weight       1
                    :alist       '(r7 r9 r2 r11)))

(setf  r9  (make-polygon    :vlist        '(v20 v19 v15 v16)
                    :slope        27.6
```

```
                        :orientation  135
                        :weight       1
                        :alist       '(r8 r13 r3 r12)))

(setf r10   (make-polygon    :vlist         '(v24 v4 v1 v21)
                        :slope        23.1
                        :orientation  045
                        :weight       1
                        :alist       '(r37 r11 r7 r4)))

(setf r11   (make-polygon    :vlist         '(v4 v20 v17 v1)
                        :slope        17.2
                        :orientation  045
                        :weight       1
                        :alist       '(r10 r12 r8 r5)))

(setf r12   (make-polygon    :vlist         '(v20 v16 v13 v17)
                        :slope        27.6
                        :orientation  045
                        :weight       1
                        :alist       '(r11 r13 r9 r6)))

(setf r13   (make-polygon    :vlist         '(v16 v15 v14 v13)
                        :slope        0.0
                        :orientation  0
                        :weight       1
                        :alist       '(r9 r6 r3 r12)))

(setf r14   (make-polygon    :vlist         '(v36 v32 v29 v33)
                        :slope        15.7
                        :orientation  0
                        :weight       1
                        :alist       '(r35 r15 r26 r18)))

(setf r15   (make-polygon    :vlist         '(v32 v8 v5 v29)
                        :slope        0.0
                        :orientation  0
                        :weight       1
                        :alist       '(r14 r16 r19 r27)))

(setf r16   (make-polygon    :vlist         '(v8 v28 v25 v5)
                        :slope        18.1
                        :orientation  0
                        :weight       1
                        :alist       '(r15 r17 r20 r28)))

(setf r17   (make-polygon    :vlist         '(v28 v10 v9 v25)
                        :slope        27.3
                        :orientation  0
                        :weight       1
                        :alist       '(r16 r30 r21 r29)))

(setf r18   (make-polygon    :vlist         '(v35 v31 v32 v36)
                        :slope        15.7
                        :orientation  090
```

```
                    :weight       1
                    :alist       '(r36 r19 r22 r14)))

(setf r19   (make-polygon    :vlist        '(v32 v31 v7 v8)
                    :slope        0.0
                    :orientation  090
                    :weight       1
                    :alist       '(r18 r20 r23 r15)))

(setf r20   (make-polygon    :vlist        '(v7 v27 v28 v8)
                    :slope        18.1
                    :orientation  090
                    :weight       1
                    :alist       '(r19 r21 r24 r16)))


(setf r21   (make-polygon    :vlist        '(v27 v12 v10 v28)
                    :slope        27.3
                    :orientation  090
                    :weight       1
                    :alist       '(r20 r30 r25 r17)))

(setf r22   (make-polygon    :vlist        '(v35 v34 v30 v31)
                    :slope        15.7
                    :orientation  180
                    :weight       1
                    :alist       '(r31 r23 r18 r26)))

(setf r23   (make-polygon    :vlist        '(v31 v30 v6 v7)
                    :slope        0.0
                    :orientation  180
                    :weight       1
                    :alist       '(r22 r24 r27 r19)))

(setf r24   (make-polygon    :vlist        '(v27 v7 v6 v26)
                    :slope        18.1
                    :orientation  180
                    :weight       1
                    :alist       '(r23 r25 r20 r28)))

(setf r25   (make-polygon    :vlist        '(v12 v27 v26 v11)
                    :slope        27.3
                    :orientation  180
                    :weight       1
                    :alist       '(r24 r30 r21 r29)))

(setf r26   (make-polygon    :vlist        '(v30 v34 v33 v29)
                    :slope        15.7
                    :orientation  270
                    :weight       1
                    :alist       '(r33 r27 r22 r14)))

(setf r27   (make-polygon    :vlist        '(v6 v30 v29 v5)
                    :slope        0
                    :orientation  270
```

```
                        :weight       1
                        :alist       '(r26 r28 r23 r15)))

(setf r28  (make-polygon    :vlist        '(v26 v6 v5 v25)
                        :slope        18.1
                        :orientation  270
                        :weight       1
                        :alist       '(r27 r29 r24 r16)))

(setf r29  (make-polygon    :vlist        '(v11 v26 v25 v9)
                        :slope        27.3
                        :orientation  270
                        :weight       1
                        :alist       '(r28 r30 r25 r17)))

(setf r30  (make-polygon    :vlist        '(v10 v12 v11 v9)
                        :slope        0.0
                        :orientation  0
                        :weight       1
                        :alist       '(r21 r25 r29 r17)))

(setf r31  (make-polygon    :vlist        '(v45 v37 v38 v39 v34 v35)
                        :slope        0.0
                        :orientation  0
                        :weight       1
                        :alist       '(r22 r36 r32)))

(setf r32  (make-polygon    :vlist        '(v34 v39 v40 v22 v23)
                        :slope        0.0
                        :orientation  0
                        :weight       1
                        :alist       '(r31 r33 r1 r38)))

(setf r33  (make-polygon    :vlist        '(v34 v23 v33)
                        :slope        0.0
                        :orientation  0
                        :weight       1
                        :alist       '(r26 r32 r34)))

(setf r34  (make-polygon    :vlist        '(v24 v33 v23)
                        :slope        0.0
                        :orientation  0
                        :weight       1
                        :alist       '(r7 r33 r35)))

(setf r35  (make-polygon    :vlist        '(v36 v33 v24)
                        :slope        0.0
                        :orientation  0
                        :weight       1
                        :alist       '(r14 r34 r37)))

(setf r36  (make-polygon    :vlist        '(v44 v45 v35 v36 v43)
                        :slope        0.0
                        :orientation  0
                        :weight       1
                        :alist        '(r31 r18 r37)))

(setf r37  (make-polygon    :vlist        '(v43 v36 v24 v21 v42)
                        :slope         0.0
                        :orientation   0
                        :weight        1
                        :alist        '(r36 r35 r10 r38)))

(setf r38  (make-polygon    :vlist        '(v42 v21 v22 v40 v41)
                        :slope         0.0
                        :orientation   0
                        :weight        1
                        :alist        '(r32 r4 r37)))
```

```
;******************************************************************
;
;  Filename:  map3.cl
;
;  By:  HILTON, Cary A.
;
;  Date:  May 91
;
;  Function:  Contains map data.
;
;             1.  defines vertices, edges, and polygons within
;                 the map boundaries
;             2.  specifies which vertices and edges each polygon
;                 is made from
;             3.  contains adjacency list for polygons
;
;             4.  map3 is identical to map2 except regions
;                 are weighted
;
;*****************************************************************

;  This map closely resembles the map used
;   by Ron Ross.

; ****************************************************************


(defstruct edge
  pt1
  pt2)

(defstruct polygon
  vlist
  slope
  orientation
  weight
  alist)

;*****************************************************************

;  Global vertex list

(setf *global-vertex-list*  '(v1 v2 v3 v4 v5 v6 v7 v8 v9
                       v10 v11 v12 v13 v14 v15 v16 v17 v18 v19
                       v20 v21 v22 v23 v24 v25 v26 v27 v28 v29
                       v30 v31 v32 v33 v34 v35 v36 v37 v38 v39
                       v40 v41 v42 v43 v44 v45))

;*****************************************************************

;  Global polygon list

(setf *global-region-list*  '(r1 r2 r3 r4 r5 r6 r7 r8 r9
                       r10 r11 r12 r13 r14 r15 r16 r17 r18 r19
                       r20 r21 r22 r23 r24 r25 r26 r27 r28 r29
```

```
                           r30 r31 r32 r33 r34 r35 r36 r37 r38))
(setf gl *global-region-list*)

;*****************************************************************

(setf  v1 '(352.0  96.0 14.5))
(setf  v2 '(480.0 224.0 14.5))
(setf  v3 '(400.0 304.0 14.5))
(setf  v4 '(272.0 176.0 14.5))

(setf  v5 '(240.0 320.0  9.0))
(setf  v6 '(240.0 448.0  9.0))
(setf  v7 '(112.0 448.0  9.0))
(setf  v8 '(112.0 320.0  9.0))

(setf  v9 '(192.0 368.0 27.75))
(setf v10 '(160.0 368.0 27.75))
(setf v11 '(192.0 400.0 27.75))
(setf v12 '(160.0 400.0 27.75))

(setf v13 '(352.0 160.0 35.75))
(setf v14 '(416.0 224.0 35.75))
(setf v15 '(400.0 240.0 35.75))
(setf v16 '(336.0 176.0 35.75))

(setf v17 '(352.0 112.0 18.0))
(setf v18 '(464.0 224.0 18.0))
(setf v19 '(400.0 288.0 18.0))
(setf v20 '(288.0 176.0 18.0))

(setf v21 '(352.0  48.0  0.0))
(setf v22 '(528.0 224.0  0.0))
(setf v23 '(400.0 352.0  0.0))
(setf v24 '(224.0 176.0  0.0))

(setf v25 '(208.0 352.0 19.5))
(setf v26 '(208.0 416.0 19.5))
(setf v27 '(144.0 416.0 19.5))
(setf v28 '(144.0 352.0 19.5))

(setf v29 '(256.0 304.0  9.0))
(setf v30 '(256.0 464.0  9.0))
(setf v31 '( 96.0 464.0  9.0))
(setf v32 '( 96.0 304.0  9.0))

(setf v33 '(288.0 272.0  0.0))
(setf v34 '(288.0 496.0  0.0))
(setf v35 '( 64.0 496.0  0.0))
(setf v36 '( 64.0 272.0  0.0))

(setf v37 '(  0.0 790.0  0.0))
(setf v38 '(*x-max* 790.0  0.0))
(setf v39 '(*x-max* 496.0  0.0))
(setf v40 '(*x-max* 224.0  0.0))
(setf v41 '(*x-max*   0.0  0.0))
```

```
(setf v42 '(352.0   0.0  0.0))
(setf v43 '( 64.0   0.0  0.0))
(setf v44 '(  0.0 0.0 0.0))
(setf v45 '(  0.0 496.0  0.0))


(setf  r1  (make-polygon    :vlist        '(v3 v23 v22 v2)
                  :slope        23.1
                  :orientation  225
                  :weight       6
                  :alist       '(r32 r2 r7 r4)))


(setf  r2  (make-polygon    :vlist        '(v19 v3 v2 v18)
                  :slope        17.2
                  :orientation  225
                  :weight       4
                  :alist       '(r1 r3 r8 r5)))

(setf  r3  (make-polygon    :vlist        '(v15 v19 v18 v14)
                  :slope        27.6
                  :orientation  225
                  :weight       8
                  :alist       '(r13 r2 r6 r9)))


(setf  r4  (make-polygon    :vlist        '(v21 v1 v2 v22)
                  :slope        23.1
                  :orientation  315
                  :weight       6
                  :alist       '(r1 r38 r10 r5)))

(setf  r5  (make-polygon    :vlist        '(v1 v17 v18 v2)
                  :slope        17.2
                  :orientation  315
                  :weight       4
                  :alist       '(r4 r6 r2 r11)))

(setf  r6  (make-polygon    :vlist        '(v17 v13 v14 v18)
                  :slope        27.6
                  :orientation  315
                  :weight       1
                  :alist       '(r5 r13 r3 r12)))

(setf  r7  (make-polygon    :vlist        '(v24 v23 v3 v4)
                  :slope        23.1
                  :orientation  135
                  :weight       6
                  :alist       '(r34 r8 r1 r10)))

(setf  r8  (make-polygon    :vlist        '(v4 v3 v19 v20)
                  :slope        17.2
                  :orientation  135
                  :weight       4
                  :alist       '(r7 r9 r2 r11)))
```

```
(setf  r9  (make-polygon    :vlist        '(v20 v19 v15 v16)
                    :slope          27.6
                    :orientation  135
                    :weight         8
                    :alist        '(r8 r13 r3 r12)))

(setf r10  (make-polygon    :vlist        '(v24 v4 v1 v21)
                    :slope          23.1
                    :orientation  045
                    :weight         6
                    :alist        '(r37 r11 r7 r4)))

(setf r11  (make-polygon    :vlist        '(v4 v20 v17 v1)
                    :slope          17.2
                    :orientation  045
                    :weight         4
                    :alist        '(r10 r12 r8 r5)))

(setf r12  (make-polygon    :vlist        '(v20 v16 v13 v17)
                    :slope          27.6
                    :orientation  045
                    :weight         8
                    :alist        '(r11 r13 r9 r6)))

(setf r13  (make-polygon    :vlist        '(v16 v15 v14 v13)
                    :slope          0.0
                    :orientation  0
                    :weight         1
                    :alist        '(r9 r6 r3 r12)))

(setf r14  (make-polygon    :vlist        '(v36 v32 v29 v33)
                    :slope          15.7
                    :orientation  0
                    :weight         4
                    :alist        '(r35 r15 r26 r18)))

(setf r15  (make-polygon    :vlist        '(v32 v8 v5 v29)
                    :slope          0.0
                    :orientation  0
                    :weight         1
                    :alist        '(r14 r16 r19 r27)))

(setf r16  (make-polygon    :vlist        '(v8 v28 v25 v5)
                    :slope          18.1
                    :orientation  0
                    :weight         4
                    :alist        '(r15 r17 r20 r28)))

(setf r17  (make-polygon    :vlist        '(v28 v10 v9 v25)
                    :slope          27.3
                    :orientation  0
                    :weight         8
                    :alist        '(r16 r30 r21 r29)))
```

```
(setf r18  (make-polygon    :vlist         '(v35 v31 v32 v36)
                     :slope          15.7
                     :orientation  090
                     :weight         4
                     :alist        '(r36 r19 r22 r14)))

(setf r19  (make-polygon    :vlist         '(v32 v31 v7 v8)
                     :slope          0.0
                     :orientation  090
                     :weight         1
                     :alist        '(r18 r20 r23 r15)))

(setf r20  (make-polygon    :vlist         '(v7 v27 v28 v8)
                     :slope          18.1
                     :orientation  090
                     :weight         4
                     :alist        '(r19 r21 r24 r16)))


(setf r21  (make-polygon    :vlist         '(v27 v12 v10 v28)
                     :slope          27.3
                     :orientation  090
                     :weight         8
                     :alist        '(r20 r30 r25 r17)))

(setf r22  (make-polygon    :vlist         '(v35 v34 v30 v31)
                     :slope          15.7
                     :orientation  180
                     :weight         4
                     :alist        '(r31 r23 r18 r26)))

(setf r23  (make-polygon    :vlist         '(v31 v30 v6 v7)
                     :slope          0.0
                     :orientation  180
                     :weight         1
                     :alist        '(r22 r24 r27 r19)))

(setf r24  (make-polygon    :vlist         '(v27 v7 v6 v26)
                     :slope          18.1
                     :orientation  180
                     :weight         4
                     :alist        '(r23 r25 r20 r28)))

(setf r25  (make-polygon    :vlist         '(v12 v27 v26 v11)
                     :slope          27.3
                     :orientation  180
                     :weight         8
                     :alist        '(r24 r30 r21 r29)))

(setf r26  (make-polygon    :vlist         '(v30 v34 v33 v29)
                     :slope          15.7
                     :orientation  270
                     :weight         4
                     :alist        '(r33 r27 r22 r14)))
```

```
(setf r27  (make-polygon    :vlist        '(v6 v30 v29 v5)
                    :slope        0
                    :orientation  270
                    :weight       1
                    :alist       '(r26 r28 r23 r15)))

(setf r28  (make-polygon    :vlist        '(v26 v6 v5 v25)
                    :slope        18.1
                    :orientation  270
                    :weight       4
                    :alist       '(r27 r29 r24 r16)))

(setf r29  (make-polygon    :vlist        '(v11 v26 v25 v9)
                    :slope        27.3
                    :orientation  270
                    :weight       8
                    :alist       '(r28 r30 r25 r17)))

(setf r30  (make-polygon    :vlist        '(v10 v12 v11 v9)
                    :slope        0.0
                    :orientation  0
                    :weight       1
                    :alist       '(r21 r25 r29 r17)))

(setf r31  (make-polygon    :vlist        '(v45 v37 v38 v39 v34 v35)
                    :slope        0.0
                    :orientation  0
                    :weight       1
                    :alist       '(r22 r36 r32)))

(setf r32  (make-polygon    :vlist        '(v34 v39 v40 v22 v23)
                    :slope        0.0
                    :orientation  0
                    :weight       1
                    :alist       '(r31 r33 r1 r38)))

(setf r33  (make-polygon    :vlist        '(v34 v23 v33)
                    :slope        0.0
                    :orientation  0
                    :weight       1
                    :alist       '(r26 r32 r34)))

(setf r34  (make-polygon    :vlist        '(v24 v33 v23)
                    :slope        0.0
                    :orientation  0
                    :weight       1
                    :alist       '(r7 r33 r35)))

(setf r35  (make-polygon    :vlist        '(v36 v33 v24)
                    :slope        0.0
                    :orientation  0
                    :weight       1
                    :alist       '(r14 r34 r37)))

(setf r36  (make-polygon    :vlist        '(v44 v45 v35 v36 v43)
                     :slope        0.0
                     :orientation  0
                     :weight       1
                     :alist       '(r31 r18 r37)))

(setf r37  (make-polygon    :vlist        '(v43 v36 v24 v21 v42)
                     :slope        0.0
                     :orientation  0
                     :weight       1
                     :alist       '(r36 r35 r10 r38)))

(setf r38  (make-polygon    :vlist        '(v42 v21 v22 v40 v41)
                     :slope        0.0
                     :orientation  0
                     :weight       1
                     :alist       '(r32 r4 r37)))
```

INITIAL DISTRIBUTION LIST

1. Defense Technical Information Center 2
   Cameron Station
   Alexandria, VA 22304-6145

2. Library, Code 52 2
   Naval Postgraduate School
   Monterey, CA 93943-5002

3. Man-Tak Shing, Code CS/Sh 4
   Department of Computer Science
   Naval Postgraduate School
   Monterey, CA 93943

4. Neil C. Rowe, Code CS/Rp 2
   Department of Computer Science
   Naval Postgraduate School
   Monterey, CA 93943

5. Robert B. McGhee, Code CS/Mz 1
   Department of Computer Science
   Naval Postgraduate School
   Monterey, CA 93943

6. CPT Cary A Hilton, Jr 1
   Route 2, Box 124A
   Mendenhall, MS 39114

7. MAJ Mark R. Kindl 1
   AIRMICS
   115 O'Keefe Building
   Georgia Institute of Technology
   Atlanta, GA 30332

Thesis
H53385 Hilton
c.1 A stochastic approach to solving the 2½ dimensional weighted region problem.